Serverless Security Primer

Top Risks and How to Mitigate

The move to serverless has made some things better and some things worse, but pretty much nothing has stayed the same. In this eBook, we'll cover the security advantages of serverless, then examine the top 6 security challenges serverless raises, along with solutions for mitigation.

Serverless

INCREASES SPEED, LOWERS COSTS

The Evolution of the Cloud

There are many benefits to moving to a serverless architecture. With automated, nearly infinite scaling, you're absolved of the need to make decisions on spinning up servers. Very little stands between developers and deployed code, enabling serverless to facilitate a shift from DevOps to NoOps. This lack of friction is valuable, speeding time to market and making it easier to maintain and test individual functions. Finally, you pay only for what you use, resulting in lower costs.

On Premise	Data center Hosting	VMs	Containers	Serverless
Application	Application	Application	Application	Application
Configuration	Configuration	Configuration	Configuration	Configuration
Scalability	Scalability	Scalability	Scalability	Scalability
Monitoring	Monitoring	Monitoring	Monitoring	Monitoring
Patching	Patching	Patching	Patching	Patching
Setup	Setup	Setup	Setup	Setup
OS	OS	OS	OS	OS
Provisioning	Provisioning	Provisioning	Provisioning	Provisioning
Virtualization	Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers	Servers
Network	Network	Network	Network	Network
Data Center	Data Center	Data Center	Data Center	Data Center
1990	2000	2000-...	2013-...	2016-...

The Security Impacts of Serverless

There are many ways in which serverless alters your security needs. In some regards, the nature of serverless improves security. So we'll start with the positives.

No More Need to Patch Servers

Historically, many exploits have been successful becausepatches haven't been updated. Serverless entirely removes that risk. With no servers, there's no longer a need to patch and secure your servers independently.

AWS, Microsoft, and Google have, for the most part, proven very reliable in keeping their parts of the stack patched andsecured, so giving them a bigger chunk of the stack certainly improves things on that end.

Attackers' Lives Become Harder

The ephemeral, stateless nature of serverless compute means that exploits don't necessarily turn into persistent presence inside your system. Lambda functions run in magical containers that self-destruct every few minutes (mostly), and the function calls themselves typically have short timeouts, so gaining a foothold in such a container will not give your attacker the prize it might have in the past.

Easily Examine The Skeleton

The fact that your application is now structured as a large number of small functions in the cloud enables you to see each unit of compute as a separate entity. This provides a fantastic opportunity for security.

Application security tools often go to incredible lengths to analyze and instrument your packaged app just to be able to observe or filter the internal flow of the app. With serverless, the bone structure and nervous system of your application are visible in the cloud deployment. There's no need to break up an app to do security inside it.

Granular Control Of Permissions

With serverless, you assign security policies to individual functions. This enables granular control over what processes and databases each function can trigger and access.

Your Apps Have Gone Serverless.

HAS YOUR SECURITY?

m

Overview

Not owning the platform means not being able to leverage the platform for security in ways you might have in the past. The threats to your serverless applications are, in many ways, the same as they were before serverless. But they may not look and act the same way. Maintaining control and security requires a paradigm shift in your thinking.

SECURITY CHALLENGES AND SOLUTIONS FOR MITIGATION

While serverless reduces some of the top security threats, vulnerabilities within the application layer remain the same, and some security threats are becoming even worse. In fact, it's a different set of issues. We'll discuss the problems related to serverless computing, common misapplication, and some proposed solutions for mitigation.

1 Loss of the Perimeter

Overview

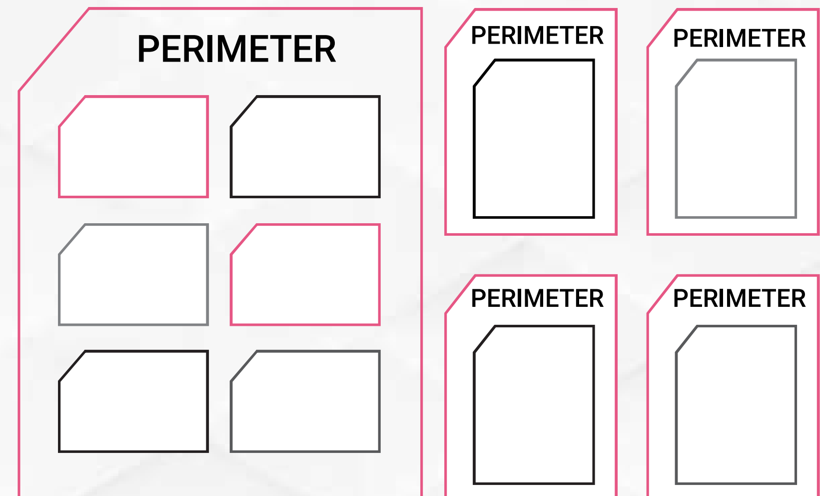
When we think of a web application exposing a lot of functionality that is all tied together, our first point of defense is typically the perimeter. We'll throw various forms of input sanitization at that boundary, scrutinizing those HTTP requests coming in. This is often achieved by using Web Application Firewalls (WAFs) or regular perimeter devices filtering the traffic.

Serverless, with it's event driven architecture means that your functions can be executed for a myriad of reasons, from diverse sources. In effect, every function is on its own and therefore has its own perimeter. Even more challenging is the fact that many of those triggers don't come from HTTP requests. That means traditional filtering solutions won't help us.

Mitigation:

Keep using your WAF and API Gateway, but be prepared to apply perimeter security at the function level as well.

A FUNCTION IS A PERIMETER THAT NEEDS TO BE SECURED



How Is Serverless Different?

The fragmentation of your application to smaller components that are callable, coupled with the use of triggers from diverse sources (such as storage, message queues, and databases) means attackers have more targets and more attack vectors.

2 Managing Permissions

Overview

Application security starts with permissions. The first goal of the security owner in any application is ensuring that the privilege level is always as low as possible. That will ensure that even if an attacker finds a way through your perimeter, they will be hard pressed to get at your valuable assets.

This is always challenging, but serverless can substantially increase the number of resources that can act and be acted upon. You must consider the policies governing the interaction between hundreds of resources, with hundreds of possible permissions in each direction.

Serverless applications can present an almost endless set of opportunities misconfigure permissions, and there are a lot of forces pulling your application in that direction.

How Is Serverless Different?

The large number of serverless resources that interact with each other requires much more effort to configure correctly. Even then permissions are correct during deployment, small changes in server configuration could suddenly open your application up to attack.

Mitigation:

Spend time crafting suitable, minimal roles for each of your functions. Review the code and configuration of each function, and enumerate all the possible actions taken by your code. Consider emerging technologies that can help craft these policies for you, and alert you any time things change.

3 Vulnerable Application Dependencies

Overview

At first glance, Serverless functions might appear to be your code— but that's not entirely true. Functions often include dependencies, pulled in from npm (Node.js), PyPI (Python), Maven (Java) or other relevant repositories. These code packages are like little pieces of foreign infrastructure embedded inside your app.

Application dependencies are similar to the oft-exploited server dependencies. They are prevalent, downloaded billions of times a month; it's hard to track which packages you're using; and they are frequently vulnerable, with new vulnerabilities disclosed regularly. Attackers are already exploiting vulnerable application dependencies, but denied the easy path of vulnerable server dependencies, they will shift to attacking these similar entities in full force.

In July 2017, a security bug bounty hunter from Moscow detailed how he was able to gain direct push rights on 73,983 NPM Packages – that's a total of 14% of the NPM ecosystem, the largest application dependency repository in the world. Because of the way many NPM packages rely on other packages, this actually meant that a whopping 54% of NPM packages could be infected with malware. This means that attackers can fairly easily get their code into your functions without directly attacking your deployment.

How Is Serverless Different?

While the issue of third party dependencies and their vulnerabilities is not new or unique to serverless, the fact that your code is spread out over a much larger set of small services, each of which imports its own set of libraries, makes managing this problem manually particularly challenging in serverless.

Conversely, the subdivision of the application into smaller services, sometimes referred to as nano-services, presents the opportunity to apply a more fine grained degree of least privilege, which can significantly limit the impact of a vulnerable library.

Mitigation:

Known vulnerabilities are as easily discoverable for the customer as they are for attackers. Securing application dependencies requires access to a good database and automated tools to continuously prevent new vulnerable packages from being used and getting alerted for newly disclosed issues.

Ensuring proper segmentation of the application into disparate services, and the scrupulous application of the principle of least privilege, can help minimize the impact of vulnerably libraries on your deployment.

4 Bad Code

Overview

Let's face it, even the best developers make mistakes, and not all of your developers are the best developers. Mistakes can easily turn into security holes, so finding a way to exterminate these bugs early is critical.

Serverless deployments, with their diverse triggers and infinite scaling can even mean that the smallest of errors can quickly turn into a self-inflicted denial-of-service attack from within your application.

How Is Serverless Different?

Serverless means there is less standing between developers and productio n, which means there are fewer places to catch mistakes in code and configuration. Additionally, with a more exposed attack surface, bugs can more easily turn into security liabilities.

Mitigation:

Training is critical. Code reviews will help as well. Mostly, though, monitor your code and configuration using tools to test configuration. Additionally, ensure that each of your functions executes with the smallest viable set of privileges, so the damage from any holes that slip through is minimal.

5 Denial-of-Service Denial-of-Wallet

Overview

Serverless provides infinite scaling, right? Not quite. Under the hood, there are various limits that the cloud providers impose on your applications. Those limits can be about maximum concurrency for your application, but also about how quickly the concurrency can rise. You may have negotiated higher limits for your application, but it's worth remembering that scaling is not infinite, so you are still susceptible to Denial-of-Service attacks if an attacker can saturate your limits.

At the same time, if you have set your concurrency limits high enough to avoid this issue, you now face the more modern fear of Denial-of-Wallet attacks, which is simply an attacker saying: "If I can't overwhelm you, I can dig deep into your pockets." If an attacker can generate 10,000 requests per second on you application and sustain that for 24 hours, the costs just for your API Gateway and Lambda Invocations could be over $5000, just for that day. If they can generate 100,000, and keep at it for a week… well you can do the math.

How Is Serverless Different?

It can be easier to enable massive auto-scaling of your service when using serverless, which can help mitigate some DoS attacks, but opens you up to DoW attacks that may not have really been relevant previously. Furthermore, you might not even realize you're under attack for a while if your app can scale enough.

Mitigation:

Deploying proper mitigations for DoS attacks, such as Amazon's API Gateway, in front of web endpoints can still help, but you must consider DoS and DoW via other triggers, such as Kinesis, and S3. Function self-protection can help detect these attacks, minimize their impacts, and dynamically adjust scaling choices to help mitigate.

6 Container Reuse

Overview

Cloud providers run your functions inside a container of some sort. Because the cost of getting your function ready to process a request is not insignificant (the "cold start") the providers will try to keep this container available to use immediately ("warm") for as long as makes sense. This means that your container could persist for a while, and that a successful attack on a container could do more damage than you might expect. Don't be fooled by the ephemeral promise, containers, and therefore attacks, can persist for hours, and maybe even days. Worse still, you can expect the cloud providers to keep containers for applications that are relatively active warm for longer and longer, as they try to provide lower latency and better performance.

How Is Serverless Different?

Well, that's the thing, it isn't as different as you have been made to expect. Yes, things are more ephemeral than containers and VMs, and we're unlikely to see function instances hanging around for months, like our VMs do. But if you're relying on this for security, you are in for a rude awakening.

Mitigation:

Consider strategies to limit the lifetime of your containers (for example, on some platforms there are APIs to make a container refresh). Some security solutions can do this for you. Furthermore, make sure you have a security solution that can detect things that try to hang around in your containers, like extra code, hidden native process, etc., and then flush those things out of the container.

Conclusion

Serverless both improves security in some ways, and generates new weaknesses and vulnerabilities. As a paradigm shift in web application structure, serverless requires a paradigm shift in security.

Like any facet of cyber security, securing your serverless application requires a variety of tactics throughout your entire application development life cycle and supply chain. Stringent adherence to best practices will improve your security posture. However, proper development is not enough. To achieve ideal protection, you must leverage tools that provide continuous security assurance, attack prevention and detection, and deception.

1. **LOSS OF THE PERIMETER**
   Keep using your WAF and API Gateway, but be prepared to apply perimeter security at the function level as well.

2. **MANAGING PERMISSIONS**
   Spend time crafting suitable, minimal roles for each of your functions. Consider emerging technologies that can help craft these policies for you, and alert you of changes.

3. **VULNERABLE APPLICATION DEPENDENCIES**
   Securing application dependencies requires access to a good database and automated tools. Help minimize the impact of vulnerable libraries by ensuring proper segmentation of the application into disparate services, and scrupulously applying least privilege.

4. **BAD CODE**
   Training is critical. Code reviews will help as well. Mostly, though, monitor your code and configuration using tools to test configuration.

5. **DENIAL-OF-SERVICE OR DENIAL-OF-WALLET**
   Deploying proper mitigations for DoS attacks, such as Amazon's API Gateway, in front of web endpoints can still help, but you must consider other triggers. Function self-protection can help detect, minimize, and mitigate these attacks.

6. **CONTAINER REUSING**
   Consider strategies to limit the lifetime of your containers. Make sure you have a security solution that can detect, and then flush out things that try to hang around in your containers.

Through continuous security assurance, attack prevention and detection, and deception the CloudGuards serverless solution helps organizations achieve control over the security of their applications. CloudGuard continually scans your infrastructure and, using machine learning, adapts your security posture to maximize protection. As a single tool doing active defense, CloudGuard gains information to use in aggregate, resulting in a unique security synergy.