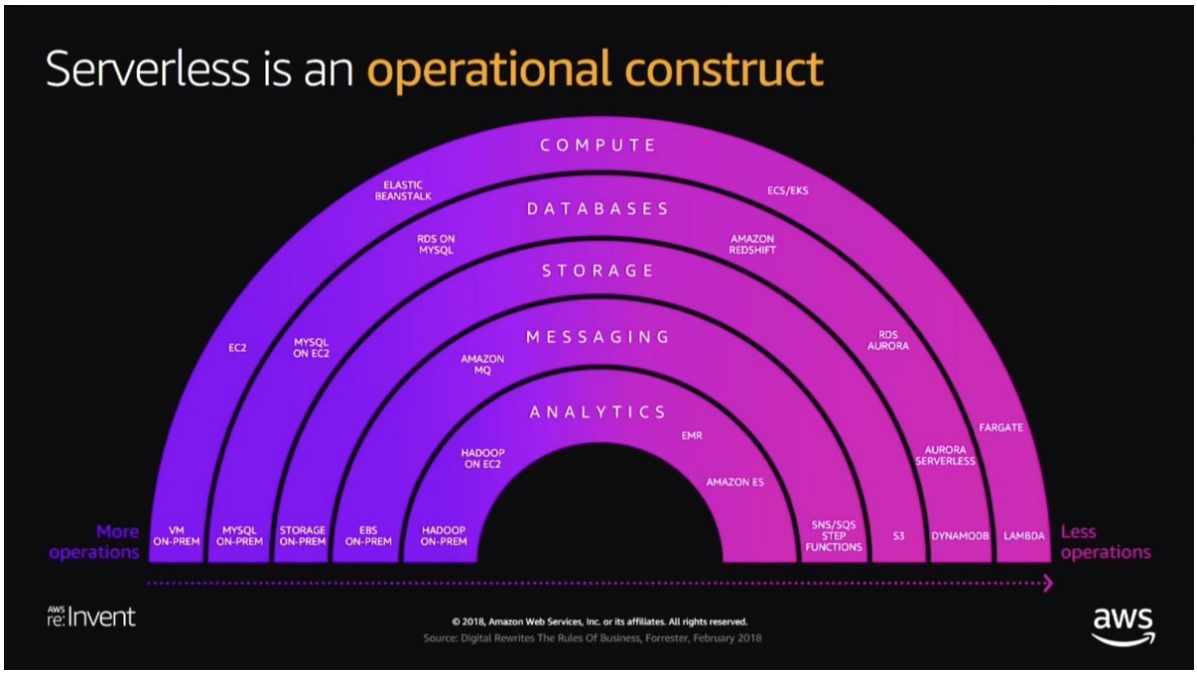
Securing Lambda Functions

By keith / February 17, 2019



First a definition.

A lambda function is a service provided by aws that runs code for you without the introducing the complexity of provisioning servers of managing Operating Systems. It belongs in a category of architectures called serverless architectures.

There's a whole slew of folks trying to define with is serverless, but my favorite definition is this.

> Serverless means No Server Ops

They're the final frontier of compute, where the idea is that developers just write code, while allowing AWS (or Google/MSFT) to take care of everything else. This includes H/W management, OS Patching, even application level maintenance like Webserver upgrades are not your problem anymore with serverless.

Nothing runs on fairy-dust though, serverless still has servers — but in this world those servers, their operating systems, and the underlying runtime (e.g. Python, Node, JVM) are fully managed services that you pay per use.

As a developer you write some code into a function. Upload that function to AWS — and now you can invoke this function over and over again without worrying about servers, operating systems or run-time.

But how does AWS achieve this?

Before we can understand how to secure a serverless function, we need to at least have a fair understanding of how Serverless functions (like AWS Lambda) work.

So how does a lambda function work?

AWS Lambda: Under the hood

This talk from re:Invent 2018 is an amazing intro into how lambda's work, and you need to watch it (if you haven't already).

Under the hood, every time a Lambda function is invoked, a container is spun-up in an AWS data center, with the memory & compute capacity specified. The container is short-lived and exits after its task is complete or has exceeded its per-specified timeout.

That container runs in a VM running Amazon Linux (a derivative of RHEL6). The container itself has a few bits and you'd expect from a standard linux install, but not all of them. For example, it has OpenSSL but not Git, Wget or Curl. For a sub-set of what's available in the container view this gist here.

Go a bit deeper, and that the container has `/bin/bash` shell, and it's environment variables contain AWS credentials that are associated with the IAM role the lambda was assigned. This is important, so I'll repeat it, an AWS function is a container with an IAM role, and the credentials to that role exists in the environment variables of the functions container.

This container interacts with all other components via the AWS API. There is nothing special — in fact, because lambdas have no persistent storage, it's only method of communicating with anything once invoked is via the network interface.

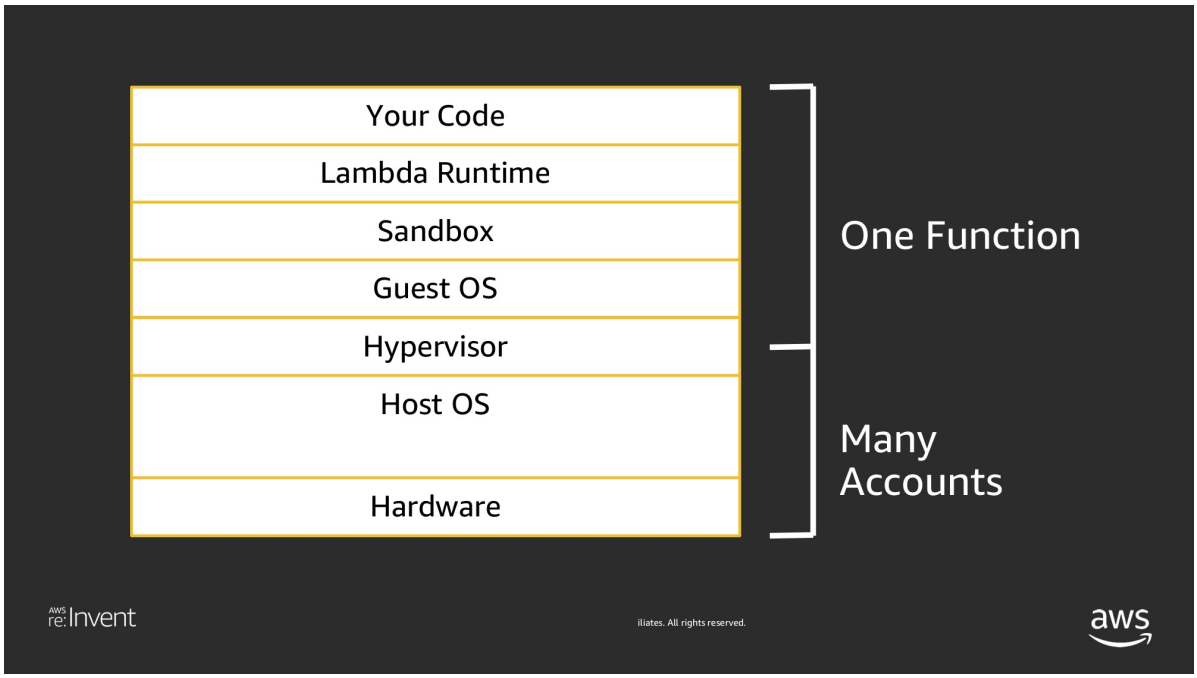
The only difference is the AWS credentials used for the calls are short-lived and automatically provisioned into the container at build-time. AWS rotates the credentials, but I'm unsure of the frequency of rotation.

Now while the container will terminate once the function has completed execution — it will not entirely disappear. AWS keeps 'warmed-up' containers on standby just in case the function is invoked again. This speeds up the next invocation, avoiding a 'cold-start' problem.

AWS will invoke the exact same container, on the exact same VM, if it has a chance to do so. This drastically improves the start-time of the function.

It's important to understand this concept, as containers have disk space available in the `/tmp` directory. A recycled container will persist the data in `/tmp`even though it might be a different invocation, lambda will freeze connections and variables outside your main handler, which some folks use to improve performance, by re-using DB connections.

A re-used container will bring the contents of the `/tmp` directory and any variables outside of your handler function `N` invocation, so if you're writing data to disk in a lambda function, it's wise to wipe it clean before exiting. Even if not for security, you're still limiting the disk-space available to the next invocation. A cool trick I learnt is to use things like Pythons ByteIO write data from memory directly to files in S3 Buckets, never touching the disk on lambda.



Also, AWS guarantee that the function is isolated from other functions at an OS level.No two functions (from even the same account) will run on the same VM, let alone two functions from different accounts. This provides a big level of isolation between functions even within the same account.

Finally, by default Lambda Functions have access to the internet. If you want to limit their connectivity you'll have to deploy them onto a VPC.

Now let's focus on Security.

Security on Lambda Function

Let's start at the end — this madman intentionally published Shell Access to his lambda function online, and even a GUI for write shell commands directly into his AWS account (lambdashell.com). It's been there since Oct-2018 (making it nearly 4 months now) and no one has managed to cause any serious damage.

Sure, some folks managed to delete his webpage using over-privileged functions, and others managed to mine some crypto (albeit at 3 seconds per invocation) — but this only happened after he intentionally introduced some loose permissions to his setup.

With default settings, even with an RCE vulnerability on a function has limited damage potential. The most serious exploit came from denial of wallet attacks, where attackers were creating thousands of Cloudwatch logs, or filling them up with useless garbage.

Lambda functions aren't perfect, but they have much smaller attack surface than EC2 instances. With limited memory, space and run time — it's much harder to leverage a potential lambda vulnerability than something similar on an EC2. A shell on EC2 would be mining crypto-currency all day long and could probably run things like wget and curl to run more payloads.

So here's a few quick tips to hardened up the Lambda function:

De-serialization & Vulnerable depedencies

bcrypt	==3.1.5	3.1.6	outdated
cffi	==1.11.5	1.12.1	outdated
pycparser	==2.19	2.19	up-to-date
six	==1.12.0	1.12.0	up-to-date

The most popular way of getting an Remote Code Execution on a lambda function is via Deserialization attacks or vulnerable dependencies. Neither of these are exclusive to serverless architectures of course, but avoid using things like Pickle (Python), serialize (Java/PHP), Marshal (Ruby) or `eval()` and you've covered ~80% of the RCE potential of your code.

And maybe you don't use pickle, but you may use Numpy that uses Pickle. Just last month, a vulnerability in numpy was found over the way it used Pickle. So be clear over what dependencies your code runs on, and what further dependencies those introduce. For python, something as simple as having a free service like GitHub or Pyup scan your requirements.txt files can alert you to these issues.

I prefer to keep dependencies such as external packages (like requests) and binaries (like Git & ffmpeg) into layers. Layers make dependencies much easier to manage, and provide a easy way to report on which functions use which dependencies.

Just these two things help significantly reduce the exposure a lambda has.

Lock down IAM & Resource

Each lambda function has an IAM role, the credentials for which are in the container it runs on. If an attacker retrieves the credentials out of the function, they can use those credentials for everything the role permits.

If the function had access to write to your DynamoDB, or read sensitive files on S3, bad things will happen.

Avoiding de-serialization and vulnerable dependencies allow us to reduce the likelihood of losing those credentials in the first place, but we should also limit each function to just the permissions it needs. This is a bit of a chore, but with plugins for frameworks like serverless-iam-roles-per-function it's a one-time high value operation.

This way, a vulnerability in one function can't assume the role of another, and it's likely you'll have many IAM roles, each with limited value, spread across the application, making any one vulnerable function less damaging.

The alternative is of one giant IAM role shared by all functions, making a vulnerability in any function result in serious damage, regardless of what the function does.

Also make sure your other AWS resources are locked down. Lambdas don't exist in a vacuum, they typically have /namoDB, S3 buckets etc, ensure those resources (especially S3) are set to private as much as possible, and accessible only by your account. Why bother with hacking lambda if the S3 bucket is exposed to the internet?

Finally, sticking with the principle of least privilege, lambda functions have access to the internet by default. If that's not needed, deploying them onto a locked-down VPC improves security even more, but this security step is quite tedious for most, a lambda in a VPC does introduce some complexity.

But that complexity ensures the attacker can't retrieve code from the internet, but has to inject it via the parameters passed to the function — hopefully this will make them easier to sanitize or detect.

Monitoring Logs & Detection

Lambda's by default come with Cloudwatch logs, that log high level stats of the function, and anything additional you've logged from your code. Unfortunately, the logs don't provide OS level details like CPU utilization or shell commands (since this is serverless), but they do provide maximum memory usage and duration of the function.

So in the scenario where an attacker has managed to manipulate your function, monitoring logs is probably the best way to detect any compromise.

And now, with cloudwatch insights which lets you run pseudo-SQL queries on all your logs it's even easier to pick for further analysis. The best thing about insights is that you don't need to download the logs, the query is run 'in-place' while the logs are still on AWS. The bad news it that the queries cost money, and are directly proportional to the size of your logs you're querying.

If you suddenly see a spike of lambda functions running longer than they should, or using more memory than usual — or even just odd exceptions that are showing up in your logs (stderr is captured onto logs), it might be time to take some hard manual analysis

Unknown unknowns

Of course, all of this is premised on the traditional view of, attack a web-app, get a shell, priv-esc or laterally move across the network.

It's kinda hard to laterally move when the function kills itself every few seconds *(lambda has a 15 minute limit, but API Gateway imposes a 29-second limit)*, and even harder to priv-esc when the containers run on the latest OS and kernel version (AWS manages this, not you!)

But maybe in this new serverless world, new attack paradigms will emerge. If data exist in S3, RDS or even Cloudwatch logs, why bother getting a shell if you can access that data via some other means. Lambda's also persist data both in `/tmp` and variables outside the handler — there's definitely an attack vector there, made worse by the fact that few know it even exists.

It helps that AWS offer VM level isolation across functions, but that's based on firecracker which is still relatively new.

It'll be exciting to see what emerges over the next 2-3 years as more applications move to serverless, and it starts becoming main-stream enough to merit the attention of real-world attackers, and not just researchers.

Conclusion

It's just a good time to be serverless.

Honorable mention

Puresec has a product called FunctionShield, which is 'sort-off' AWS plugin that allows you limit the functionality of the lambda function. The 'plugins' are language specific but are usually just imported packages, that allow you to code statements in function that prevent the function from writing to `/tmp`, accessing the internet, and even disable child process execution.

Because it's a statement into your function, you have very fine-grained control over what you code can do where — i.e you can limit writing to `/tmp`only at specific points of your code, so that an attacker gaining RCE in the earlier points of your code has less to leverage on.

It's a great idea, but I'm no so certain that the solution to securing lambda is adding one more imported package/dependency into your code. That being said, this does give you flexibility above and beyond what most other solution offer, plus the solution logs data to cloudwatch which helps in monitoring and detection.

Other resources:

- OWASP Serverless Top 10
- Serverless Goat (similar to WebGoat)
- Damn Vulnerable Serverless App (similar to DVWA)
- Introduction to Deserialization
- LambDash (shell in a Lambda Function)
- FunctionShield (locking down lambdas)
- Newbies guide to securing lambda

- LambdaShell.com (lambda shell exposed to the internet)
- Attacking lambdashell (write up ; lambda shell)
- AWS Lambda Under the Hood (video from re:invent 2018)
- Peek behind the curtains on serverless platforms

ADD COMMENT

ASTOUND US WITH YOUR INTELLIGENCE

Enter your comment here...

READ MORE

Access Keys in AWS Lambda

June 14, 2020

Contact Tracing Apps: they're OK.

May 10, 2020

Sharding SQS

May 10, 2020

Logging within AWS Lambda Functions (python edition)

April 30, 2020