See the latest version of this whitepaper on : https://threatresponse.cloud

Presenter Bios

Andrew Krug is a Security Engineer for Mozilla
Graham Jones is a Software Developer at LegitScript

Hacking Serverless Runtimes

Serverless technology is getting increasingly ubiquitous in the enterprise and startup communities. As micro-services multiply and single purpose services grow, how do you audit and defend serverless runtimes? The advantages of serverless runtimes are clear: increased agility, ease of use, and ephemerality (i.e., not managing a fleet of “pet” servers). There is a trade off for that convenience though: reduced transparency. We will deep dive into both public data and information unearthed by our research to give you the full story on serverless, how it works, and attack chains in the serverless cloud(s) Azure, AWS, and a few other sandboxes. Who will be the victor in the great sandbox showdown?

Serverless Runtimes -- How are they Different?

Many people think that serverless runtimes are simply “the cloud” rebranded. This is only partially true. Serverless is the embodiment of PaaS or Platform as a Service. Serverless creates unique problems when it comes to application security and the lower cost of deploying using serverless will cause usage to increase over the next several years.

Serverless Makes Development Easy

In serverless application design, applications are broken apart into smaller units of code (so-called “microservices”) that deploy into individual sandboxes. In the case of web applications these functions are chained to an API Gateway connecting parameters and REST routes to those chunks of code. Each time a route is called the function spins up, performs its small job, and returns. Developing code and getting it deployed with serverless technology can be very easy. Additionally, it largely removes operational problems of scaling applications across multiple servers.

Serverless Makes Development Hard

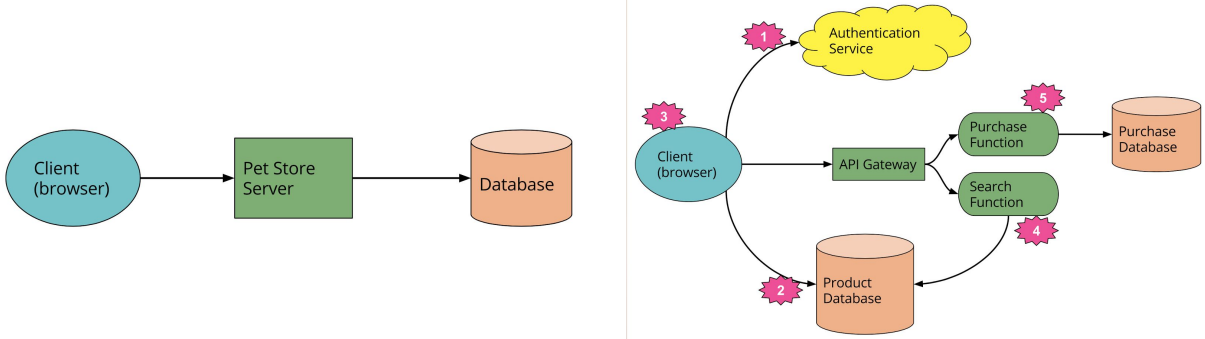
In the old development model we had a monolithic application and there were known ways to ensure the security of that application. In serverless the model of application design is quite different and getting telemetry on testing and execution is quite difficult.

The Great Sandbox Showdown : An attack and defense surface analysis of serverless runtimes.

Andrew Krug and Graham Jones

Figure 1.1

Traditional Application Model **New Application Model**



Reference 1.(Fowler 2017)

Fig 1.1 above shows a traditional application (left) and a serverless application (right). In our traditional application the architecture a 3 tier full stack architecture. In the serverless app each block of code becomes a single responsibility function chained to an API Gateway and a 3rd party Identity management solution.

Distributed application models such as the one in figure 1.1 have several challenges when addressing application security. At RSAC 2017 Signal Sciences Researcher, James Wicket cited four key areas of concern. These are not dissimilar to traditional development but have specific focuses when applied to sandbox computing.

Key Areas of AppSec Focus Ref. (Wicket 2017)	
Software Supply Chain	How does the software get vetted? Is security unit testing easy or hard to do? **Can we use traditional tools?**
Delivery Pipeline Security	How does the software get deployed and **what controls determine what code is executing**?
Data Flow Security	Where is the data going? Is it the right data? **Am I paying to exfiltrate my own secure dataset?**
Attack Detection	Since the container is truly “ephemeral” how do I detect anomalous behavior? **How do I respond?**

These four areas fall into two categories:

Category 1: What happens before the code is deployed?
Category 2: What happens after the code is deployed?

Pre-Code Deploy Challenges

In a pre serverless era your AppSec technology was likely well established. The flow went something like:

- Developer Writes Code
- Developer Writes Tests
- OpSec Writes Security Tests
- Code is Committed and Pushed Somewhere
- More Tests Run (Unit, Function, Integration)
- Code is accepted or rejected based on the output of the tests

Serverless has its own challenges in that very standard pipeline. Integration and functional testing becomes difficult. This is due to the fact that there are often abstraction layers that simply can not run locally for developers. You can't just spin up your own API Gateway at home to learn to secure it. The code must be deployed for a complete integration test in many cases, which makes integration testing difficult. Some frameworks like Chalice, Zappa, and serverless.js make this easier by running local web servers to replace routers, but there is no standardized way to perform integration or security tests.

The old adage that high-friction security will have low impact holds true here. *Developers moving to serverless for the first time will likely skip types of testing that are difficult to perform.* Serverless is not a magic bullet and despite smaller attack surfaces, "Bad Code is Bad Code".

Post-Code Deploy Challenges

There have been some great research projects to try and help developers and security operations alike understand how to defend the serverless runtimes. In 2016 the creator of Zappa Framework known as "Miserlou" showcased at CCC a demonstration of a worst-case exfiltration attack on a serverless application. The attack used a combination of permissions that were too broad and insecure default permissions to use AWS Tags and CloudWatch log groups to exfiltrate data. (4. Miserlou 2017)

In this worst case scenario how do you defend yourself? How do you detect anomalies in serverless runtimes? If there is production code in Lambda, Azure, Webtask or the like without a mature log and telemetry output pipeline it increases the risk of a breach that can not be detected or traced.

The Great Sandbox Showdown : An attack and defense surface analysis of serverless runtimes.

Andrew Krug and Graham Jones

The Importance of Profiling Runtimes

Currently there are a number of code sandboxes for running serverless code. AWS Lambda, Microsoft Azure Functions, and Webtask are just a few of the popular frameworks available. We as consumers are not given a lot of information about how these runtimes work under the hood which creates an inherent risk. History has shown us all as a community that closed architectures that rely on secrecy are not inherently secure architectures. There is little public documentation about each runtime leaving us far from full knowledge of the environment setup.

Lambda What We Know

(http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html)

Amazon Web Services to their credit published a “Lambda: How it Works” article. Combining this and our profiling information, we now know the following:

- Lambda is Containers
- Runs on Amazon Linux (RHEL 6 derivative)
- Uses a single IAM role to determine permissions within ecosystem
- Only has internet egress if:
    - Not spun up in a VPC
    - Connected to an internet gateway
- Lambda has a writable /tmp directory
- /tmp provides transient cache that can be used for multiple invocations.
- Processes or callbacks initiated by your Lambda function that did not complete when the function ended resume if AWS Lambda chooses to reuse the container.

Azure What We Know

(https://github.com/projectkudu/kudu/wiki)

Project Kudu publishes plenty of interesting information about the general Azure Functions system, though some specifics are left unknown. Some key differences from Lambda:

- The section of the Azure App that the Function data is stored and run in is writable
- The C:\ drive, which is separate, is still not writable.
- Functions are deployed within an App
- Cannot make calls to `Get-WMIObject`.
- Can query the list of event logs on the system, but can’t access any of them.
- Does have general internet egress (as opposed to a VPC lambda).
- Has significant permissions within its App

While Project Kudu is related to Microsoft, being under the umbrella of the .NET Foundation, they are not technically part of the Azure team.

Webtask.io

(https://webtask.io/docs/how)

Webtask wins the award for greatest architecture transparency. They have published a full briefing on exactly what technologies they use to power Webtask. Here are the main points:

- They have two Webtask clusters deployed in different AWS regions (us-west-1 and us-east-1). One of them is the primary cluster, and the other is a failover cluster. They are using a failover routing policy at the Amazon Route 53 level to implement that failover logic.
- Each Webtask cluster consists of a 3 VM deployment with Elastic Load Balancing (ELB) in front of the cluster. ELB has configured health check monitoring and is able to take VMs out of circulation upon failure.
- The VM placement across several Availability Zones in an AWS region supports high availability at the cluster level.
- Webtasks are built on top of CoreOS, Docker, etcd, and fleet. (Janczuk 2017)

Our Mission and Research Project

Due to lack of vendor transparency in the sandbox space we elected to create a set of libraries that would execute system level calls to mine the environment for information. After all, every good data scientist knows that “you can’t manage what you don’t measure.”

Project Goals

1. Gain sufficient data to compare security features across serverless runtimes.
2. Gather telemetry in a universal and exportable way for analysis
3. Propose and attempt a model for the detection of anomalies in serverless environments
4. Create and automate a response plan for AWS Lambda Compromises as a blueprint for the community.

This area intentionally left blank

Andrew Krug and Graham Jones

Our Analysis Pipeline

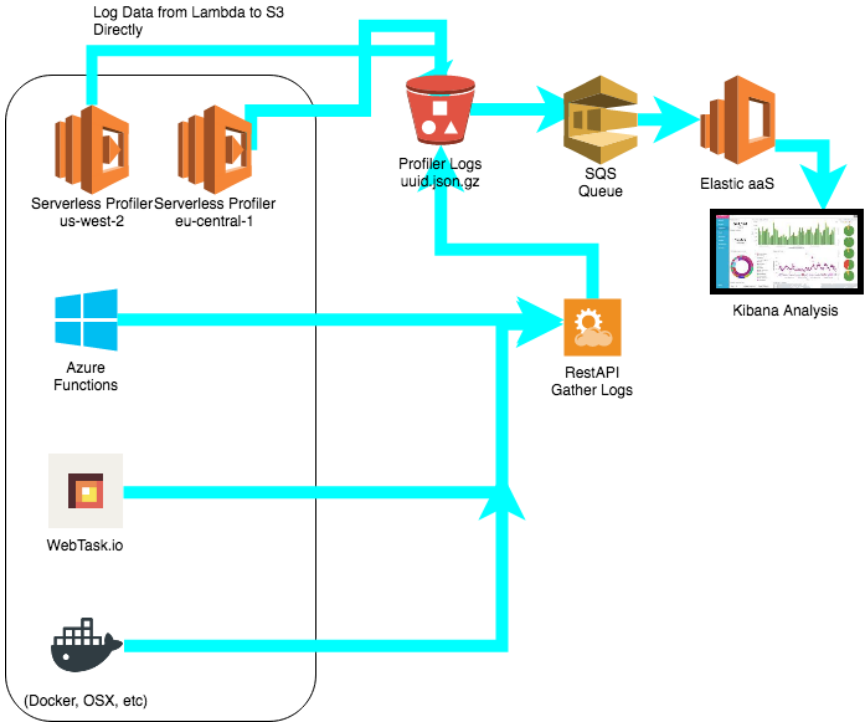


Figure 2.1 The ThreatResponse lambda profiler pipeline is a simple telemetry exporter than can be included in any lambda function. The output is JSON which is either posted to a REST API that gathers data or is deposited directly to S3. Put operations in S3 trigger an automatic event to insert the output into a message queue for ingestion into elastic using FluentD.

Our Profilers

The profilers were developed primarily in python and ported to nodejs for inclusion in as many serverless runtimes as possible. The profilers mine the operating system for information using the “os” and “platform” modules as well as shell outs for anything that cannot be natively gathered in language. For compactness they depend only on what is in the language natively without the use of libraries.

Release 1.0 Features:

- Gather ‘/etc/issue’
- Gather Present Working Directory

- System Version Information
- Telemetry on Attached Filesystems
- Writability and Persist Ability
- Warmness Checks (Is my provider recycling my sandbox?)
- Processor and Memory Telemetry
- Information on Native Libraries in Runtime
- Running Process
- Contents of Environment
- Sensitive Environment Identification and Sanitization
- Hashing of suspicious files in tmp locations

Companion Sample Code

Profilers: https://github.com/ThreatResponse/python-lambda-inspector
Serverless Observatory: https://github.com/ThreatResponse/serverless-observatory

Detection Pipeline Concepts

The larger the attack surface of the runtime, the more telemetry you'll need to check. For comparison here is an infographic comparing initial relative sizes of profiler outputs. Smaller means less attack surface and therefore less to monitor.

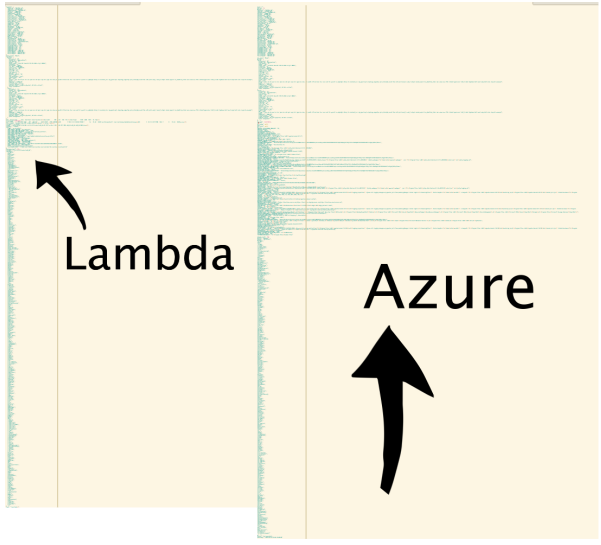


Figure 3.1 - (Left) shows preliminary comparisons early in this research of relative runtime data that's present for mining. The profile of Azure for example was quite a bit larger due to the proliferation of environment variables. Everything from Tomcat 7-9 to IIS seemed present in the base image.

Areas of Focus for IOCs (Indicators of Compromise)

Serverless functions have essentially two paths of compromise. Code delivery or code execution. In code delivery an attacker compromises and corrupts the code and that "evil" code manages to makes it way into unsuspecting production via the deployment pipeline. In an

injection scenario an attacker is taking advantage of a classic type of attack like RCE or SQLi for example to cause the code to behave in an unintended way.

Deployment Pipeline

- Deployments during unscheduled time periods.
- Many deployments in a short period of time.
- Updates to code outside of the purview of the CI System

Injection (RCE, SQLi, etc)

- Execution times outside of “normalcy” for the function.
- Lambda handler (“the main”) is overridden or has garbage on the end
- Datasets attempting to leave API gateway that are flagged confidential
- Non standard cloudwatch group name
- Non standard tags
- Strange files across warm executions

Response to Serverless Incidents (AWS Specific)

When bad code goes bad how do you stop the bleeding? Whether it’s CI pipeline corruption or an RCE the procedure is likely the same. We want to stop execution. This can be done in effectively two steps.

Step 1 - Revoke any STS Tokens that could have been granted using the lambda functions execution role. There is actually a button for this now in the console.

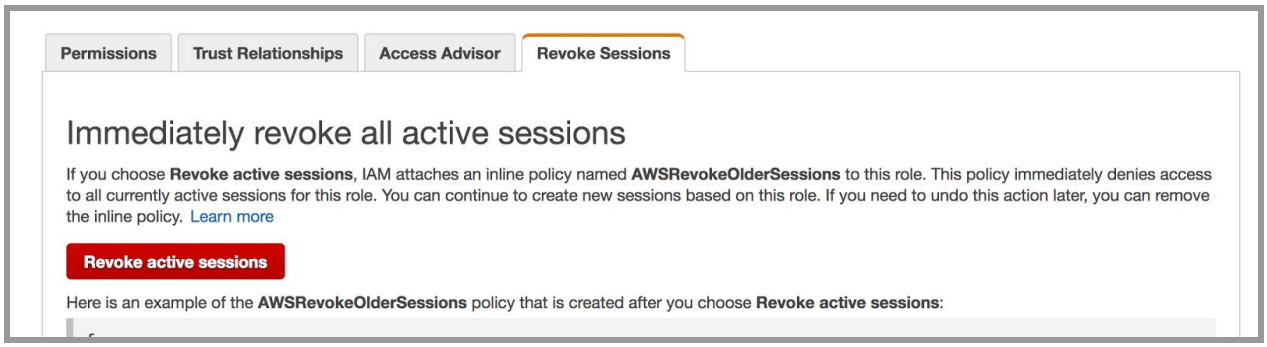


Figure 4.1 shows the button in the IAM Console that suspends any temporary session tokens for the role. Effectively drawing a hard line in the sand to prevent lateral movement and persistence using stolen credentials.

This area intentionally left blank.

Step 2 - Revoke Execution of the function by applying a deny IAM policy.

```
# Sample deny all IAM Policy
  {
   "Effect": "Deny",
   "Action": [
    "*"
   ],
  }
```

The example IAM policy can be pre-created in the account and attached to any lambda execution role. Because in IAM policy, Deny clauses are higher priority than Allow, any subsequent executions will receive access denied.

Examples of pivoting after RCE

We've created vulnerable apps in AWS Lambda and Azure Functions to demonstrate how some of these possible weaknesses could be detected by an attacker and some of what can be achieved after gaining arbitrary code execution. While the vulnerabilities in these apps are largely trivial, the focus is on the attacker's capabilities that may not be readily apparent.

Lambda: How's your IAM?
The lambda app is a slackbot that, when triggered, will dump git changelogs back into the slack channel. It contains a string concatenation vulnerability that lets the requester run arbitrary shell commands. Since the lambda container does not restrict lambdas from spinning up arbitrary processes, this translates to being able to run arbitrary code through the interpreter of our choice.

In this specific implementation, we use our arbitrary execution to download a payload from the internet in the form of a python script and then execute it. The script simply gathers the IAM permissions of the lambda and then POSTs them back out for further study by the attacker. While there is no specific damage done here, this is the kind of information gathering that can be done to inform further, more damaging attacks.

This area intentionally left blank

Depending on what the IAM permissions allow, a huge number of more damaging attacks are possible. Even something seemingly benign like a framework's built-in permission to S3 can lead to sensitive data being leaked or deleted. Before adapting a serverless framework, it's important to audit what security implications you are signing up for.

```
# Subset of default IAM policy by the Zappa serverless framework
 {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::*",
            "Effect": "Allow"
  }
```

Code for the lambda vulnerable app is available at https://github.com/ThreatResponse/poor-webhook.

Azure: How are your functions grouped?
In AWS, all Lambdas within an account are siblings, though none have any sort of access to each other unless explicitly provided by IAM. In Azure, though, Functions are grouped within Applications - more or less a single Windows server. This leads to several unexpected ramifications for someone unfamiliar with the details of the architecture.

Functions within an Application share a disk partition as well as having the same general system settings (.NET Framework version, environment variables, etc). Metering and restriction of resource usage is also done on a per-Application basis (6. Kudu 2017). What may not be readily apparent, given serverless systems' emphasis on ephemerality and reusability, is the amount of persistence and access that Functions within an Application have between each other.

The Azure vulnerable application is an API that accumulates credit card charges in order to do a single round of billing at the end of the month. There is a single function available to a user via API endpoint to check their card balance. There are two other functions that are inaccessible to general users: one to return a total list of balances and one to sum up the charges and perform the billing.

In the Azure Functions nodeJS implementation, they provide a callback function to call when execution of the function is complete. Using a contrived RCE vulnerability in the card balance

endpoint, we are able to construct vulnerable payloads to do a number of surprising things. The general process is that after our arbitrary code executes getting the data we're interested in, we can then place the data we want to exfiltrate in the `context` object and trigger an end to the function immediately. This has the added advantage of not having to pass data all the way through the regular execution of the function while additionally not requiring unexpected data streams, such as if we exfiltrated data via a separate POST.

Through a series of Node.js calls of basic file operations, we first list all the Functions that exist in the Application. Once establishing that other Functions exist, we show that we can:

- Gain API access to other functions by copying over our existing API credentials into their credential files
- Read execution methods of other Functions, including data such as cron timers / API endpoints
- Change other Functions to execute in different ways (invoke a cron Function via API call)
- Dump the source of other Functions
- Edit the source of other Functions

Just like in AWS, you should be aware that a compromised Azure Function can pivot to other items in your account. However, it is even more critical to be aware that Functions within an Application have a huge amount of access to the system and to each other.

Code for the Azure vulnerable app is available at https://github.com/ThreatResponse/serverless-vulnerable-azure.

Detection of Attacks

The ecosystem of attack detection in traditional systems is fairly mature. Anomaly detection is prevalent with tools like auditd. Unfortunately, current serverless providers don't provide tools of this nature, leaving us with fewer alternatives. This section will focus primarily on AWS, but many of the philosophies are applicable to other serverless providers as well.

Especially if your workload is relatively predictable, some of the best information you can gather are the execution metrics of your functions. Large outliers in execution time, memory usage, and possibly even number of calls can be red flags. Both of the vulnerable app demos involve either enough additional processing (especially on initial payload downloading) or skipping enough of the function's execution flow to make a significant time difference.

If you can spare the in-function overhead, there is some degree of sanity checking that you can do at runtime also. Options that could detect our or other exploits include:

- Check any writable places in the filesystem to see if unexpected files exist
    - If you have files you are persisting yourself, do they have the right checksum?

- (if applicable to your architecture) Check your function's source against some separately-stored hash
- Are any processes running that you don't expect?
- Are permissions that your app has what you would expect?

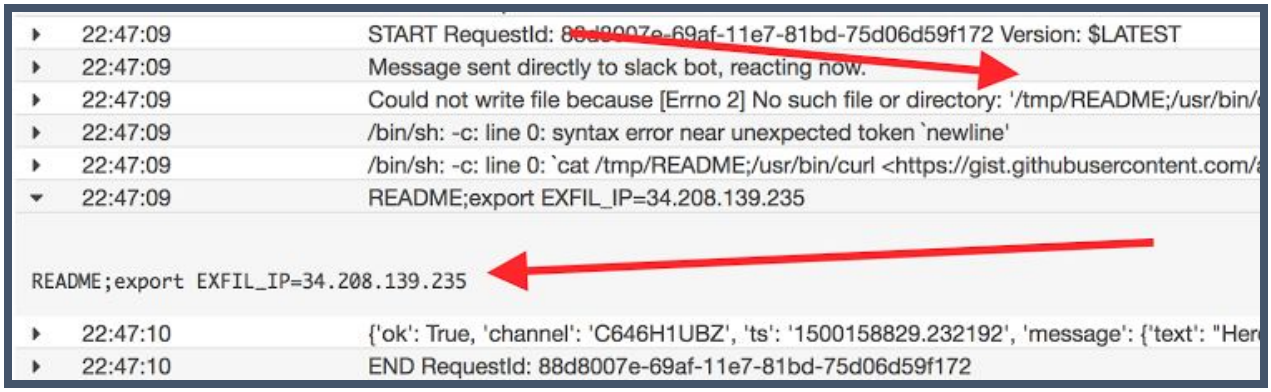
Essentially, similarly to how all of these exploit possibilities could be detected and reported by an attacker, they can be preemptively checked and reported on by a healthy process. The downside to this system is that it incurs non-trivial per-execution costs.

The integrated logging systems can provide helpful information as well. Serverless functions often don't work properly if some vulnerability is being exploited - both demo apps leave evidence in the logs that they weren't running properly. Just like traditional log analysis, being able to find this information in the logs will be based on having a good idea of what is in logs in a healthy scenario.

If the same execution of an Azure Function calls `context.done()` multiple times (as in the vulnerable app), a log entry will be displayed stating such:

```
2017-07-15T01:55:48.024 Error: 'done' has already been called.  Please check your script for extraneous calls to 'done'.
```

With the Lambda demo, we can see an example of a log that is not 'inherently wrong' like the Azure one but could be flagged as unusual:



Additionally, much of the wisdom of doing things pre-serverless still apply. It's still important to track usage of any services that serverless functions have access to, limit their credentials, etc.

Conclusion

No product is secure against an administrator setting up the wrong configuration (or vulnerable code), and features often can be security problems in different contexts.

When setting up a serverless system, it's important to understand the security choices that you're making or that your platform defaults to. Check your IAM profiles for Lambdas and make sure they use a minimal set of permissions. In Azure, be aware of the tradeoffs of placing all your Functions within a single Application. Additionally, be aware of the system that you're running on. Being aware that you're running a vulnerable runtime of Node.js or referencing old versions of modules can be a huge step towards being aware of the issues you might face.

Even though it's serverless and you don't control the systems for in-depth auditing, you still have tools available that you should make use of. Tracking logs for extraneous statements, auditing execution data, and even occasionally sanity-checking your running environment to make sure it's what you expect can go a long way towards detecting activity by attackers. Additionally, just like all your other systems, have a plan for when an intrusion does happen.

There are plenty of reasons to use serverless systems and security doesn't have to be a dealbreaker. By being aware of some of the possible attacks, indicators of compromise, and mitigation opportunities, we hope that serverless systems can be a secure and successful tool for everyone to use.

References

1. Serverless Architectures, Martin Fowler (https://martinfowler.com/articles/serverless.html) Accessed March 3, 2017
2. Gartner Top Technology Trends of 2017 (www.gartner.com/smarterwithgartner/gartners-top-10-technology-trends-2017) Accessed March 3, 2017
3. Serverless Security Are You Ready for the Future, James Wicket (http://www.slideshare.net/wickett/serverless-security-are-you-ready-for-the-future) Accessed March 3, 2017
4. Gone in 60 Milliseconds, Miserlou (https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds) Accessed January 17, 2017
5. Sandboxing in the Era of Containers, Tomasz Janczuk https://medium.com/aws-activate-startup-blog/sandboxing-code-in-the-era-of-containers-294edb3a674?adbsc=startups_20150325_42670736&adbid=580743244339752960&adbpl=tw&adbpr=168826960#.l75uulrkg Accessed March 3, 2017
6. Project Kudu (https://github.com/projectkudu/kudu) Accessed July 12, 2017