Serverless Security
What are we up against?

Atul Ahire, Harvijay Kapoor, Shanu Mandot

Atul_Ahire@McAfee.com, Harvijay_Kapoor@McAfee.com, Shanu.Mandot@Gmail.com

Abstract

Developing applications using serverless architecture helps one get relieved from the daunting task of constantly applying security patches for the underlying operating system and application servers. These tasks are now the responsibility of the serverless architecture provider. From a software development perspective, organizations adopting serverless architectures can now focus on core product functionality, and completely disregard the underlying operating system, application server or software runtime environment.

Due to these benefits, there is a lot of demand for going serverless right now. There is a community forming around these designs and major cloud service providers such as Amazon Web Services (AWS), Microsoft and Google are all pushing the concept. Large enterprises are even jumping on board.

Serverless is real, and it's here now. And yet, serverless is no silver bullet from a security perspective. Analysis of over 1,000 open-source serverless applications revealed that 21% of them have critical vulnerabilities or were misconfigured, according to security researchers (PureSec 2018). They also cited that six percent had their sensitive data, such as application program interface (API) keys and credentials, stored in publicly accessible repositories. Serverless architectures introduce a new set of issues that must be considered when securing such applications.

This paper explains some of the most prevalent security issues in serverless applications, their impact and how we can mitigate them. This papers also covers some of the tools and solutions which can be helpful to detect the security issues introduced with serverless architecture.

Biography

Atul Ahire is Software Development Engineer in Test at McAfee, with more than 7 years of experience in Software Development, Test Automation, Build Release and Deployment. His areas of interest include Cloud solution design, deployment, and DevOps.

Harvijay Kapoor is Partner integrations lead at McAfee, with 10 years of experience in Software Development. He is skilled in Project Management and handling Implementation Activities for Cloud SaaS Product Integrations. Solutioning, Client Engagement, Demos, Trainings and Post-sales Support are his main areas of strength.

Shanu Mandot is QA Lead at TechChefs, with 7 years of experience in Software QA. Her areas of interest are Security Testing and Agile methodologies.

1 Introduction

Traditionally, we have built and deployed web applications where we have some degree of control over the HTTP requests that are made to our server. Our application runs on that server and we are responsible for provisioning and managing the resources for it. There are a few issues with this approach:

a. We are charged for keeping the server up even when we are not serving out any requests.
b. We are responsible for uptime and maintenance of the server and all its resources.
c. We are also responsible for applying the appropriate security updates to the server.
d. As our usage scales we need to manage scaling up our server as well. And as a result, manage scaling it down when we don't have as much usage.

For smaller companies and individual developers this can be a lot to handle. This ends up distracting from the more important job that we have; building and maintaining the actual application. At larger organizations this is handled by the infrastructure team and usually it is not the responsibility of the individual developer. However, the processes necessary to support this can end up slowing down development times as we cannot just go ahead and build our application without working with the infrastructure team to help us get up and running. As developers we've been looking for a solution to these problems and this is where serverless computing comes in.

Serverless computing (or serverless for short), is an execution model where the serverless platform provider (like AWS, Azure, or Google Cloud) is responsible for executing a piece of code by dynamically allocating the resources. They only charge for resources used to run the code. The code is typically run inside stateless containers that can be triggered by a variety of events including http requests, database events, queuing services, monitoring alerts, file uploads, scheduled events (cron jobs), etc. The code that is sent to the cloud provider for execution is usually in the form of a function. While serverless abstracts the underlying infrastructure away from the developer, servers are still involved in executing our functions.

By its very nature, Serverless addresses some of today's biggest security concerns. By eliminating infrastructure management, it pushes its security concerns to the platform provider. Unfortunately, attackers won't simply give up, and will instead adapt to this new world. More specifically, serverless will move attacker's focus from the servers to the application and defenders should adapt priorities accordingly.

2 Serverless Overview

Let us understand the basics of serverless architecture and its importance.

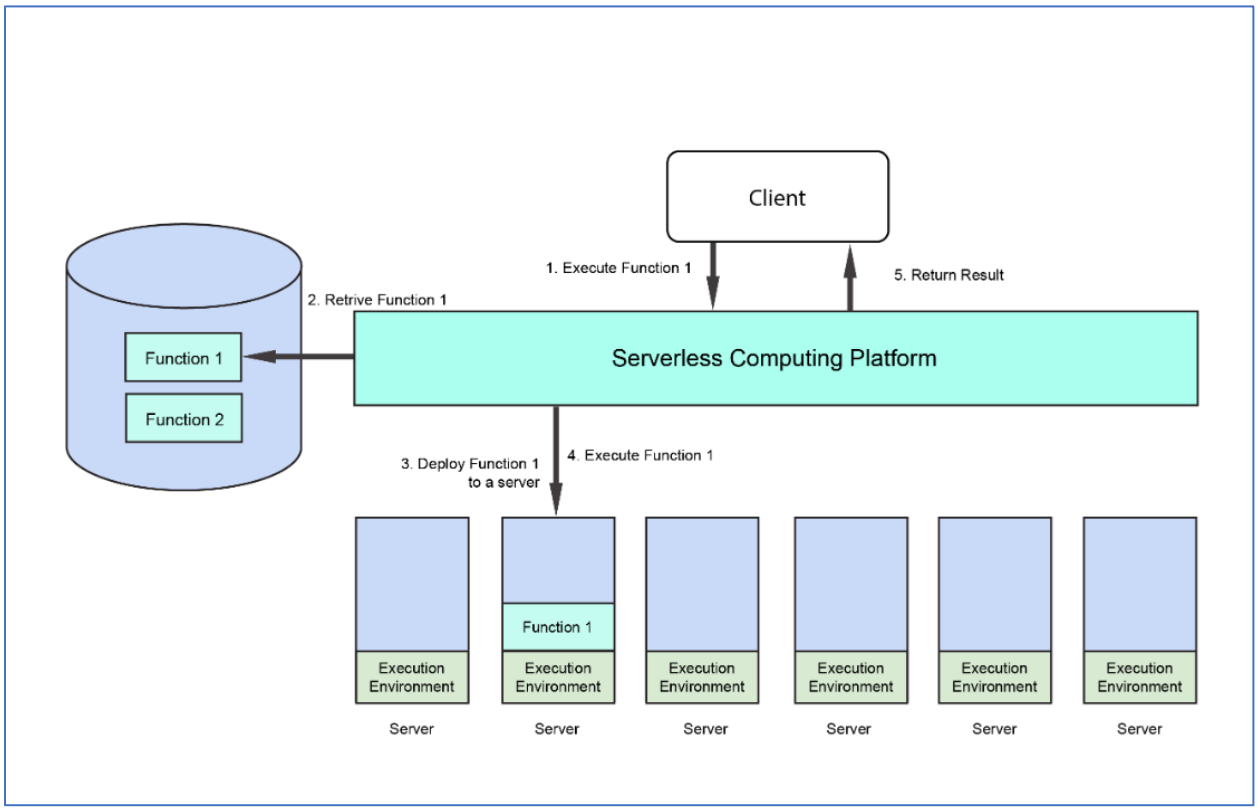
2.1 What is Serverless Computing?

Serverless computing, does not mean that we no longer use servers to host and run code; nor does it mean that operations engineers are no longer required. But consumers of serverless computing no longer need to spend time and resources on server provisioning, maintenance, updates, scaling, and capacity planning. Instead, all these tasks and capabilities are handled by a serverless platform and are completely abstracted away from the developers and IT/operations teams. As a result, developers focus on writing their applications business logic. Operations engineers can elevate their focus to more business-critical tasks.

In practice, this approach can work in two different ways:

1. Functions-as-a-Service (FaaS) typically provides event-driven computing. Developers run and manage application code with functions that are triggered by events or HTTP requests. Developers deploy small units of code to the FaaS, which are executed as needed as discrete actions, scaling without the need to manage servers or any other underlying infrastructure.

2. Backend-as-a-Service (BaaS) are third-party API-based services that replace core subsets of functionality in an application. Because those APIs are provided as a service that auto-scales and operates transparently, this appears to the developer to be serverless.



2.2 Why is Serverless Important?

Adoption of serverless delivers the following benefits:

1. **Zero Server Ops**: Serverless dramatically changes the cost model of running software applications through eliminating the overhead involved in the maintenance of server resources. Without provisioning, updating, or managing server infrastructure, a company can save significant overhead costs. In addition, since a serverless FaaS or BaaS can instantly and precisely scale to handle each individual incoming request, the serverless approach automatically scales down the compute resources so that there is never idle capacity.

2. **No Compute Cost When Idle**: One of the greatest benefits of the serverless approach from a consumer perspective is that there are no costs resulting from idle capacity. For example, serverless compute services do not charge for idle VM or containers. When there's no work being done, there are no charges being racked up.

3. **Time to production**: The serverless development model aims to radically reduce the number of steps involved in conceiving, testing, and deploying code, with the aim of moving functionality from the idea stage to the production stage in days rather than months.

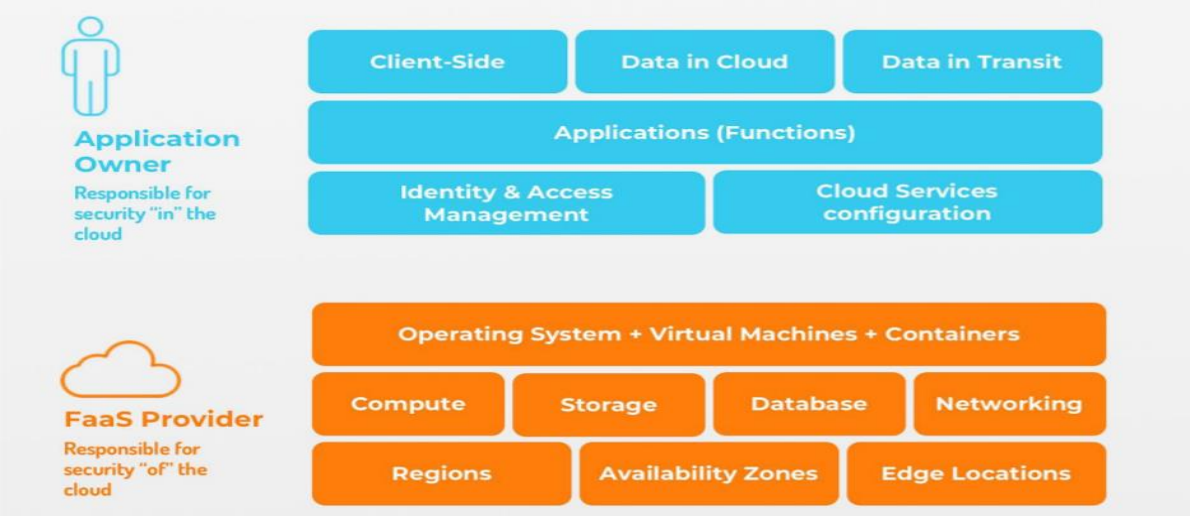
3 Security in Serverless Architecture

When you host the application in public cloud, the security responsibilities get distributed between the application owner and the public cloud service provider. Who is responsible for what depends on the cloud service model you use and is referred as shared security responsibilities model.

In serverless architectures, the serverless provider is responsible for securing the data center, network, servers, operating systems, and their configurations. However, application logic, code, data, and application-layer configurations still need to be robust and resilient to attacks. These are the responsibility of application owners.

Companies adopting serverless should ensure that they provide trainings and create awareness among their employees and help them understand the new shared responsibilities model.

The following image demonstrates the shared security responsibilities model, adapted to serverless architectures:



4 Serverless Security Challenges

Serverless architectures introduce a new set of issues that must be considered when securing such applications. These include:

4.1 Increased attack surface

Serverless functions consume data from a wide range of event sources, such as Hypertext Transfer Protocol (HTTP) application program interfaces (APIs), message queues, cloud storage, internet of things (IoT) device communications, and so forth. This diversity increases the potential surface dramatically. Many of these messages cannot be inspected by standard application layer protections, such as web application firewalls (WAFs). Example, web application firewalls (WAFs) are only capable of inspecting HTTP(s) traffic to the serverless application, but will not provide protection on any other event trigger types like cloud storage events, stream processing events, message queue events etc.

4.2 Attack surface and system complexity

Serverless design patterns call for web applications to be broken into constituent functions that are then reassembled in a serverless context. The attack surface in serverless architectures can be difficult for some

to understand given that such architectures are still somewhat new. Many software developers and architects have yet to gain enough experience with the security risks and appropriate security protections required to secure such applications. Visualizing and monitoring serverless architectures is still more complicated than standard software environments as developers must make many decisions about which functions within their applications to expose to end users and then securely and accurately do so.

As an example, when we think of authentication in traditional applications, it can be applied using single authentication provider for entire domain/app and execution of proper authentication becomes simple. However, in Serverless applications, each serverless function acts as nano-service requiring unique authentication. Moreover, cloud services used by serverless application also require unique authentication and therefore, application of proper authentication grows tremendously complex and difficult to visualize upfront.

4.3 Inadequate security testing

Performing security testing for serverless architectures is more complex than testing standard applications, especially when such applications interact with remote third-party services or with backend cloud services, such as Non-Structured Query Language (NoSQL) databases, cloud storage, or stream processing services. Additionally, existing commonly used automated scanning tools are currently not adapted to examining serverless applications.

Example of some of the commonly used scanning tools are:

- **DAST (dynamic application security testing)** tools will only provide testing coverage for HTTP interfaces. This limited capability poses a problem when testing serverless applications that consume input from non-HTTP sources or interact with backend cloud services.
- **SAST (static application security testing)** tools rely on data flow analysis, control flow, and semantic analysis to detect vulnerabilities in software. Since serverless applications contain multiple distinct functions that are stitched together using event triggers and cloud services (e.g., message queues, cloud storage or NoSQL databases), statically analyzing data flow in such scenarios is highly prone to false positives. These rule sets used in these tools will need to evolve to provide proper support for serverless applications.

5 Critical risks with Serverless Applications

While Serverless isn't inherently bad for security, its sheer scale emphasizes some very critical security risks. Maintaining control and security requires a paradigm shift in our thinking. Below are some of the most prominent security risks categories for serverless architecture and few best practices one can follow to mitigate them:

5.1 Function Event Data Injection

At a high level, injection flaws occur when untrusted input is passed directly to an interpreter before being executed or evaluated. Injection flaws in applications are one of the most common risks to date and have been thoroughly covered in many secure coding best practice guides.

In the context of serverless architectures, however, function event-data injections are not strictly limited to direct user input (such as input from a web API call). Most serverless architectures provide a multitude of event sources, which can trigger the execution of a serverless function. Examples include:

- Cloud storage events (e.g., Amazon Web Services Simple Storage Service (AWS S3), Azure Blob Storage, Google Cloud Storage)
- NoSQL database events (e.g., AWS DynamoDB, Azure Cosmos DB)

- SQL database events
- Stream processing events (e.g., AWS Kinesis)
- HTTP API calls
- IoT device telemetry signals
- Message queue events
- Short message service (SMS) notifications, push notifications, emails, etc.

This rich set of event sources increases the potential attack surface and introduces complexities when attempting to protect serverless functions against event-data injections. This is especially true because serverless architectures are not nearly as well-understood as web environments where developers know which message parts shouldn't be trusted (e.g., GET/POST parameters, HTTP headers, etc.).

The most common types of injection flaws in serverless architectures are presented below:

- Function runtime code injection (e.g., Node.js/JavaScript, Python, Java, C#, Golang) (OWSAP Code Injection 2013)
- SQL injection (OWSAP SQL Injection 2016)
- NoSQL injection (OWSAP NoSQL Injection 2016)
- Publish/Subscribe (Pub/Sub) Message Data Tampering (Security Compass Blog 2016)
- Server-Side Request Forgery (SSRF) (OWSAP SSRF 2016)

Example:

Consider a job candidate curriculum vitae (CV) filtering system, which receives emails with candidate CVs attached as Portable Document Format (PDF) files. The system transforms the PDF file into text to perform text analytics. The transformation of the PDF file into text is done using a command line utility (pdf to text). The developer of this serverless function assumes users will provide legitimate PDF file names and does not perform any sanity check on incoming file names (except for rudimentary examinations to ensure file extensions are indeed ".pdf"). The file name gets embedded directly into the shell command, and this weakness allows a malicious user to inject shell commands as part of the PDF file name.

Mitigation Steps:

- Never trust input or make any assumptions about its validity.
- Never pass user input directly to any interpreter without first validating and sanitizing it.
- Make sure code always runs with minimum privileges required to perform its task.
- If you apply threat modeling in the development lifecycle, ensure consideration of all possible event types and entry points into the system; do not assume input can only arrive from the expected event trigger.
- Inspect event data using a Serverless Security Platform (where applicable, organizations may use a web application firewall instead, to inspect incoming HTTP/HTTPS traffic to the serverless applications; note: application layer firewalls are only capable of inspecting HTTP(s) traffic and will not provide protection on any other event trigger types).

5.2 Broken Authentication

Since serverless architectures promote a microservices-oriented system design, applications built for such architectures may contain dozens (or even hundreds) of distinct serverless functions, each with a specific purpose.

These functions are weaved together and orchestrated to form the overall system logic. Some serverless functions may expose public web APIs, while others may serve as an "internal glue" between processes or other functions. Additionally, some functions may consume events of different source types, such as cloud storage events, NoSQL database events, IoT device telemetry signals or even SMS notifications.

Applying robust authentication schemes—which provide access control and protection to all relevant functions, event types, and triggers—is a complex undertaking, and can easily go awry if not executed carefully.

Example:

Imagine a serverless application which exposes a set of public APIs, all of which enforce proper authentication. At the other end of the system, the application reads files from a cloud storage service where file contents are consumed as input to specific serverless functions. If proper authentication is not applied to the cloud storage service, the system is exposing an unauthenticated rogue entry point—an element not considered during system design.

Mitigation Steps:

- Organizations should use continuous security health check facilities provided by their serverless cloud providers to monitor correct permissions and assess them against existing corporate security policies.
- As an example, Organizations using AWS infrastructure should implement AWS Config rules to continuously monitor and assess their environment. Examples of AWS Config rules include:
    - Discover newly deployed AWS Lambda functions
    - Receive notifications on changes made to existing AWS Lambda functions
    - Assess permissions and roles (Identity and Access Management (IAM)) assigned to AWS Lambda functions
    - Discover newly deployed AWS S3 buckets or changes in security policies made to existing Buckets
    - Receive notifications on unencrypted storage
    - Receive notifications on changes made to existing AWS Lambda functions
- It is recommended that we should use authentication facilities provided via the serverless environment or by the relevant runtime. For example:
    - AWS Cognito or single sign-on (SSO)
    - AWS API Gateway authorization facilities
    - Azure App Service Authentication / Authorization
    - Google Firebase Authentication
    - IBM Bluemix App ID or SSO

5.3 Over-Privileged Function Permissions and Roles

Since serverless functions usually follow microservices concepts, many serverless applications contain dozens, hundreds or even thousands of functions. Resultantly, managing function permissions and roles quickly becomes a tedious task. In such scenarios, organizations may be forced to use a single permission model or security role for all functions—essentially granting each function full access to all other system components.

When all functions share the same set of over-privileged permissions, a vulnerability in a single function can eventually escalate into a system-wide security catastrophe. Thus, a serverless function should have only the privileges essential to performing its intended logic—a principle known as “least privilege.”

Example:

Consider a function which receives data and stores it in DynamoDB table using “DynamoDB put_item()” method. While the function sole logic is to store data into database, the developer made a mistake and assigned full permissions to that function as shown below:

```
Effect: Allow
Action:
-'dynamodb:*'
Resource:
-'arn:dynamodb:us-east-1:*******:table/TABLE_NAME'
```

The appropriate, least-privileged role should have read:

```
Effect: Allow
Action:
-'dynamodb:PutItem'
Resource:
-'arn:dynamodb:us-east-1:******:table/TABLE_NAME'
```

Mitigation Steps:

- The “blast radius” from a potential attack can be contained by applying Identity and Access Management (IAM) capabilities relevant to your platform, and ensuring each function has a unique user-role (run with the least amount of privileges required to perform its task properly).
- Organizations should adopt an automated solution (such as a serverless security platform) for statically scanning serverless function code and configurations, flag over-privileged IAM roles and automatically generate least-privileged roles.

5.4 Inadequate Function Monitoring and Logging

One of the key aspects of serverless architectures is the fact that they reside in a cloud environment, outside of the organizational data center perimeter. As such, “on premise” or host-based security controls become irrelevant as a viable protection solution. This in turn, means that any processes, tools and procedures developed for security event monitoring and logging, becomes obsolete.

While many serverless architecture vendors provide extremely capable logging facilities, these logs in their basic/out-of-the-box configuration, are not always suitable for the purpose of providing a full security event audit trail. In order to achieve adequate real-time security event monitoring with the proper audit trail, serverless developers and their DevOps teams are required to stitch together logging logic that will fit their organizational needs. Many successful attacks could have been prevented if victim organizations had efficient and adequate real-time security event monitoring and logging.

There are many ways in which an attacker can exploit the fact that serverless applications lack proper application-layer logging, as an example:

- Attempts to inject malicious SQL payloads (SQL Injection) in event-data fields, which will not appear in standard cloud-provider logs.
- Attempts to send brute-force authentication requests
- Attempts to invoke functions with additional event-data fields containing malicious data

Mitigation Steps:

- Organizations adopting serverless architectures should augment log reports with serverless-specific information, such as:
    - Logging API access keys related to successful/failed logins (authentication).
    - Logging attempts to invoke serverless functions with inadequate permissions (authorizations).
    - Logging all successful/failed deployment of new serverless functions or configurations (change).

- Logging any change to function permissions or execution roles (change).
- Logging any change in function trigger definitions (change).
- Logging the outbound connections initiated by serverless functions (network).
- Logging all serverless function execution timeouts (failure reports).
- Logging concurrent serverless function execution limits once reached (failure reports).

5.5 Insecure Application Secrets Storage

As applications grow in size and complexity, there is a need to store and maintain “application secrets.” Examples include:

- API keys
- Database credentials
- Encryption keys
- Sensitive configuration settings

One of the mistakes related to application secrets storage, is to simply store these secrets in a plain text configuration file that is a part of the software project. In such cases, any user with “read” permissions on the project can get access to these secrets. The situation gets much worse if the project is stored on a public repository.

Also, in serverless applications, each function is packaged separately. A single centralized configuration file cannot be used which leads developers to use “creative” approaches such as using environment variables. While environment variables are a useful way to persist data across serverless function executions, in some cases, such environment variables can leak and reach the wrong hands.

Mitigation Steps:

- It is critical to store application secrets in a secure, encrypted storage environment and maintain encryption keys in a centralized encryption key management infrastructure or service. Most serverless architecture and cloud vendors offer such services, and also provide developers with secure APIs that can easily and seamlessly integrate into serverless environments.
- Organizations that decide to persist secrets in environment variables should always encrypt data. Decryption should only occur during function execution and by using proper encryption key management

5.6 Denial of Service and Financial Resource Exhaustion

In the past decade, denial-of- service (DoS) attacks have increased dramatically in frequency and volume. Such attacks became one of the primary risks facing nearly every company with an online presence.

While serverless architectures bring promises of automated scalability and high availability, they also come with limitations and issues which require attention.

Most serverless architecture vendors define default limits on the execution of serverless functions, such as:

- Per-execution memory allocation
- Per-execution ephemeral disk capacity
- Maximum execution duration per function
- Maximum payload size
- Per-account concurrent execution limit
- Per-function concurrent execution limit

Depending on limit and activity types, poorly designed or configured applications may result in unacceptable

levels of latency, or even render applications unusable.

As an example, an attacker may send numerous concurrent malicious requests to a serverless function which uses the package until the concurrent executions limit is reached. As a result, this will deny other users access to the application. An attacker may also push the serverless function to “over-execute” for long periods, essentially inflating the monthly bill and inflicting a financial loss on the target organization.

Mitigation Steps:

- Write efficient serverless functions that perform discrete, targeted tasks.
- Set appropriate timeout limits for serverless function execution
- Set appropriate disk usage limits for serverless functions
- Apply request throttling on API calls
- Enforce proper access controls to serverless functions

5.7 Flow Manipulation in Serverless Function execution

Business logic manipulation is a common problem in many types of software and serverless architectures. However, serverless applications are unique, as they often follow the microservices design paradigm and contain many discrete functions. These functions are chained together in a specific order, which implements the overall application logic.

In a system where, multiple functions exist - and each function may invoke another function - the order of invocation may be critical for achieving the desired logic. Moreover, the design might assume that certain functions are only invoked under specific scenarios and only by authorized invokers.

Business logic manipulation in serverless applications may also occur within a single function, where an attacker might exploit bad design or inject malicious code during the execution of a function, for example, by exploiting functions which load data from untrusted sources or compromised cloud resources.

Example:

Consider application logic for a serverless application which calculates cryptographic hash for files uploaded into a cloud storage bucket. The sequence in application logic is as follows:

- Step No. 1: A user authenticates into the system.
- Step No. 2: The user calls a dedicated file-upload API and uploads a file to a cloud storage Bucket.
- Step No. 3: The file upload API event triggers a file size sanity check on the uploaded file, expecting files with an 8 KB maximum size.
- Step No. 4: If the sanity check succeeds, a “file uploaded” notification message is published to a relevant topic in a Pub/Sub cloud messaging system.
- Step No. 5: As a result of the notification message in the Pub/Sub messaging system, a second serverless function—which performs the cryptographic hash—is executed on the relevant file.

This system design assumes functions and events are invoked in the desired order. However, a malicious user may manipulate the system in two ways:

1. If the cloud storage bucket does not enforce proper access controls, any user may be able to upload files directly into the bucket, bypassing the size sanity check (which is only applied in Step No. 3). Malicious users may upload numerous large files, essentially consuming all available system resources as defined by the system’s quota.
2. If the Pub/Sub messaging system does not enforce proper access controls on the relevant topic, any user may be able to publish numerous “file uploaded” messages—forcing the system to

continuously execute the cryptographic file hashing function until all system resources are consumed.

In both cases, an attacker may consume system resources until the defined quota is met, and then deny service from other system users. Another possible outcome is an inflated monthly bill from the serverless architecture cloud vendor (also known as “Financial Resource Exhaustion”).

Mitigation Steps:

- Protecting serverless applications against business logic manipulation can be done by leveraging serverless security platform capable of enforcing normal application flow and verifying that functions behave as designed.
- In addition, organizations should design the system without making any assumptions about legitimate invocation flow. Developers should set proper access controls and permissions for each function.

5.8 Obsolete Functions, Cloud Resources and Event Triggers

Like other types of modern software applications, over time some serverless functions and related cloud resources might become obsolete and should be decommissioned. Obsolete serverless application components may include:

- Deprecated serverless functions versions
- Serverless functions that are not relevant anymore
- Unused cloud resources (e.g. storage buckets, databases, message queues, etc.)
- Unnecessary serverless event source triggers
- Unused users, roles or identities
- Unused software dependencies

During a preliminary attack reconnaissance phase, malicious users will usually begin by mapping the serverless application, looking for the path of least resistance. Obsolete functions / cloud resources, unnecessary event source triggers or IAM roles are the most likely targets for abuse.

As an example, developers might leave event source triggers, which are unaccounted for and are probably not monitored properly. Such event triggers may enable an attacker to bypass authentication mechanisms or abuse business logic.

Mitigation Steps:

- Organizations should continuously perform discovery and pruning of obsolete serverless assets, cloud resources, IAM roles and any unknown serverless code that might have been deployed outside of the normal development process.
- Such discovery can be automated by using a cloud security posture management (CSPM) solution, or a serverless security platform (SSP).

6 Serverless Security Tools/Solutions

Following are some of the commercially available tools and solutions which can be used to secure the serverless applications:

6.1 Serverless Security Platform (SSP) by PureSec

PureSec's SSP is designed exclusively for serverless applications, and provides an end-to-end application security solution for serverless, which is tightly integrated into the CI/CD process.

The PureSec serverless security platform provides protection for applications using AWS Lambda, Azure Functions, Google Cloud Functions and IBM Cloud Functions so you can ensure that your functions are free from risk and safe from threats at every stage of the application lifecycle.

6.2 Protego

Protego is an application security platform that targets full-lifecycle security, from deployment to runtime. The web-based UI application surfaces security-focused visualizations, including the security posture explorer, third-party vulnerability reports, and policy manager.

The platform combines cloud account scanning to detect and address problems with roles and permissions; an analytics engine using machine-learning and deep-learning algorithms to detect threats, anomalies, and malicious attacks; and runtime protection that inspects and filters function-input data.

The Protego platform supports AWS, Google Cloud Platform, and Azure, and functions using Node.js, Python, and Java runtimes.

6.3 Synk

Snyk is one of the popular solutions to monitor, find and fix the vulnerabilities found in the application's dependencies. Recently, they have introduced the integration with AWS Lambda and Azure Functions which allow you to connect and check if a deployed application is vulnerable or not.

6.4 DivvyCloud

DivvyCloud, which Gartner identifies as a CSPM (Cloud Security Posture Management) solution, touches the following Cloud Management Platform categories:

- Identity, Security, and Compliance
- Monitoring and Analytics
- Inventory and Classification
- Cost Management & Resource Organization (at a peripheral level)

DivvyCloud is designed to effectively manage the perpetual shift of cloud infrastructure. By combining continuous real-time monitoring and a range of automation, along with the right cultural approach and processes, can enable an organization to solve cloud security issues around governance of multi-cloud, compliance based on a range of standards (CIS, NIST, HIPPA, etc), and security concerns tied to common misconfigurations issues.

7 Conclusion

Serverless is an exciting evolution in the world of infrastructure. It isn't inherently better or worse for security, compared to other infrastructure models, but it does change how we operate our software, and requires we adapt how we secure it. The Serverless ecosystem, including best practices and tools, is being shaped now. Serverless designs are becoming mainstream. The business advantages are undeniable. The top security challenge remains tackling the myth that "security's taken care of by our providers." Once that's shattered, there is real work to be done to secure these designs.

As adoption of the cloud continues and matures, securing serverless applications is an inevitable requirement, as many enterprises are adopting this approach to their cloud applications. The goal of security is to ensure that our data is being handled in the manner we expect. In a serverless world that means map out our data, write high quality code, trust (but verify!) what our provider is doing and monitor our application extensively. In our paper we have tried to list some of the important risks what top industry

practitioners and security researchers with vast experience in application security, cloud and serverless architectures believe and some of the best practices industries adopting serverless can use to mitigate them.

References

2018. Martin fowler's blog. "Serverless Architectures". https://martinfowler.com/articles/serverless.html (Accessed June 20, 2019).

2018. PureSec Blog BY Ory Segal. https://www.puresec.io/blog/puresec-reveals-that-21-of-open-source-serverless-applications-have-critical-vulnerabilities.

2017. Open Web Application Security Project(OWASP). "Top 10-2017 Application Security Risks". https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

2016. Security Compass Blog, "Publish-Subscribe Threat Modeling". https://blog.securitycompass.com/publish-subscribe-threat-modeling-11add54f1d07

2018. Steven J. Vaughan-Nichols. "Servers? We don't need no stinkin' servers!". Linux and Open Source Journal.

2016. Open Web Application Security Project(OWASP)."SQL Injection". https://www.owasp.org/index.php/SQL_Injection

2016. Open Web Application Security Project(OWASP)."No SQL Injection". https://www.owasp.org/index.php/Testing_for_NoSQL_injection

2016. Open Web Application Security Project(OWASP). "Server-Side Request Forgery (SSRF)". https://www.owasp.org/index.php/Server_Side_Request_Forgery

2013. Open Web Application Security Project(OWASP)."Code Injection". https://www.owasp.org/index.php/Code_Injection

2019. Cloud Security Alliance. "The 12 Most Critical Risks for Serverless Applications". https://blog.cloudsecurityalliance.org/2019/02/11/critical-risks-serverless-applications (Accessed July 14, 2019).

Amazon Web Services. "Shared Responsibility Model". https://aws.amazon.com/compliance/shared-responsibility-model (Accessed June 20, 2019).

Amazon Web Services. AWS Lambda - https://aws.amazon.com/lambda (Accessed June 15, 2019).

Google Cloud. Google Cloud Functions - https://cloud.google.com/functions (Accessed June 15, 2019).

Microsoft Azure. Azure Functions - https://azure.microsoft.com/en-in/services/functions (Accessed June 16, 2019).

Serverless Security Platform by PureSec. https://www.puresec.io/serverless-security-platform (Accessed July 20, 2019).

Protego Platform. https://www.protego.io/platform (Accessed July 20, 2019).

Open Source Security Platform by Snyk. https://snyk.io/product (Accessed July 24, 2019).

DivvyCloud:"Unified Visibility and Monitoring". https://divvycloud.com/unified-visibility-and-monitoring (Accessed August 16, 2019).