Securing Microservices on Serverless Platforms

Cassandra Young

September 16, 2019

1 Introduction

The microservices architecture is loosely defined as collection of smaller services arranged together in a workflow, often utilizing a serverless platform. Microservices architecture, as opposed to the traditional, monolithic model, utilizes these services or functions in a loosely coupled manner, often hosted separately and communicating via REST APIs or other lightweight protocols. When implemented on a serverless platform, such as AWS Lambda, this can result in an extensible application development process.

The serverless model allows the realization of a few fundamental capabilities, including transference of responsibility for infrastructure, flexible resource scaling, high availability, and use-based cost as opposed to resource-based cost. [1] On a serverless platform, the burden of running, maintaining and upgrading networks and servers, and the operating systems running on them, is shifted to the cloud provider. This allows the developer to focus on the code itself, and, as is the case with AWS, allows integration with other cloud-hosted services such as virtual compute, databases, and storage.

While serverless architecture removes the burden of securing physical devices, networks and operating systems, it relies on the shared responsibility model while representing a wider and more complex attack surface. Data integrity and security, logging and monitoring, and application code are the responsibility of the developer. [2]

Limited visibility and/or control over serverless functions in a more distributed model raises a number of security concerns, particularly when that function is presented as a "black box" with little or no documentation of what that service may connect to, and how. An investigation into the security risks of a distributed microservices architecture using AWS Lambdas has to explore the security risks of a variety of Amazon's most utilized services, the connections between them, and the levels of access involved in connecting them.

[1]Serverless Architectures with AWS Lambda (AWS)
[2]Security Overview of AWS Lambda (AWS)

2 Microservices on a Serverless Platform

2.1 AWS Lambda: Function-as-a-Service

On a serverless platform, the hardware, server and OS level are obfuscated from the customer. When invoking a Lambda, AWS manages the resources and dependencies needed for the code to run. It spins up a container which runs on an account-specific MicroVM. Importantly, "A microVM is dedicated to an AWS account, but can be reused by execution environments across functions within an account. MicroVMs are packed onto an AWS owned and managed hardware platform (Lambda Workers). Execution environments are never shared across functions, and microVMs are never shared across AWS accounts."[3]

The only writeable directory in this environment is */tmp*, with code stored and run out */var/task*. The execution context can be reused for up to 15 minutes. [4] "After a Lambda function is executed, AWS Lambda maintains the execution context for some time in anticipation of another Lambda function invocation. In effect, the service freezes the execution context after a Lambda function completes, and thaws the context for reuse, if AWS Lambda chooses to reuse the context when the Lambda function is invoked again." [5]

A Lambda can be accessed either directly via its native RESTful service API [6], or when triggered by an event such as an HTTPS request to a configured AWS API Gateway. Other triggers that can be configured to include an upload to an S3 bucket, DynamoDB database update, CloudWatch event, and others. It can be programmed to access any public AWS resource, or resource within a VPC that is configured to allow it. Access can also be explicitly granted across different accounts. To enable this, the Lambda temporarily assumes an IAM role with an attached policy that grants permissions to access the resource. This policy can grant any level of access to the resource specified, from granular read-only access, to full control.

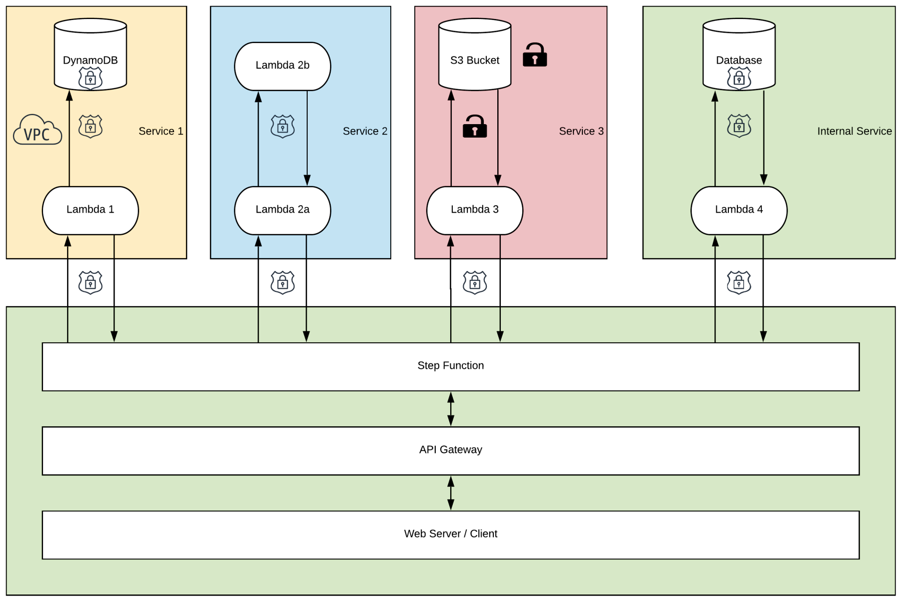
2.2 Serverless within a Distributed Development Model

In an application development process where one developer creates and configures all of the involved functions, securing that application from end to end can be accomplished in a consistent, policy-based manner. However, in a distributed development environment, that may not be possible. The developer of an application may not have any control over or visibility into all of the interconnected pieces of their application.

[3]Security Overview of AWS Lambda
[4]AWS Lambda Limits (AWS)
[5]AWS Lambda Execution Context (AWS)
[6]Serverless Architectures with AWS Lambda (AWS)

There are two areas of responsibility in this situation. The first is that of a developer to secure their own code and runtime environment, to use best practices to secure data and connections to other services, and to enable additional security options when reasonable. The second is that of a developer who is calling an external service to take steps to mitigate risk where possible.

Example reference diagram:



3 Exposure and Risks

3.1 Vulnerabilities within the Lambda runtime

Regardless of data protection in transit or at rest, any compromise of the lambda's code or environment itself can expose event data and provide opportunities for privilege escalation. Remote code execution via source code or CI/CD pipeline compromise, or through insecure code, represent well documented vulnerabilities in AWS Lambda. Unsanitized input can be leveraged to access the underlying container, escalate privileges of the applied IAM role, and exfiltrate event data. Privilege escalation can allow an attacker access to other account resources, limited only by the scope of the Lambda's assumable roles.

The Lambda context is spun up and runs in a sandboxed container running on a virtual machine. From the runtime environment, it's possible to access the underlying container. As noted previously, "each execution context provides

512 MB of additional disk space in the /tmp directory. The directory content remains when the execution context is frozen, providing transient cache that can be used for multiple invocations."[7] Testing shows that the persistence of the execution context can be leveraged during an attack to store tools and exfiltrate data.[8]

The best defense against runtime vulnerabilities is defensive coding. Code should be written with awareness of injection flaws, and should sanitize and check all input whenever possible. Developers can also consider utilizing an API Gateway, which offers built-in protection against DDoS attacks, logging and monitoring, and traffic management.

3.2 Data Exposure at Rest and In Transit

In a serverless application architecture, any number of microservices can be utilized, exposing an unknown quantity of services, and the necessary data transfers between them. Understanding the native security features of the most frequently used services is an important part of assessing risk.

The majority of AWS services default to or enforce connections over HTTPS/SSL, and utilize at rest encryption as a default. One notable exception to this is S3, which is non-encrypted by default and accepts both HTTP and HTTPS traffic.

Most popular AWS services allow for multiple options when securing data in transit and at rest, including options to utilize AWS Key Management Service when encrypting data at rest. AWS utilizes key hierarchy, which allows for symmetric key encryption at the lowest level, and provides a secure way for keys to be stored next to the data they encrypt.[9] This results in a secure but fast and efficient way for data to be decrypted when needed.

Below is a table of defaults and options for encryption in transit and at rest.

[7]AWS Lambda Execution Context
[8]Gone in 60 Milliseconds (Jones)
[9]AWS Key Management ServiceCryptographic Details

Service	In Transit	At Rest	Charges
AWS Lambda	HTTPS only (all)	RAM only (all)	N/A
via AWS API (Direct)	”	”	”
via API Gateway (Direct)	”	”	”
Step Function	HTTPS only	N/A	N/A
DynamoDB	HTTP or HTTPS *		
AWS owned CMK (Default)	Default: HTTPS	”	N/A
AWS managed CMK	”	”	KMS charges
Client-side Encryption	”	”	N/A
S3 Buckets	HTTP or HTTPS		
Default	”	unencrypted	N/A
SSE-S3	”	S3-managed CMK	KMS charges
SSE-KMS	”	KMS-managed CMK	KMS charges
SSE-C	HTTPS only	Client-provided key	N/A

* FIPS endpoints in DynamoDB are HTTPS only

While encrypting data in transit and at rest provides additional security, it does not eliminate the possibility of exposure to an attacker who has gained access to the privileges necessary to decrypt it.

3.3 Privilege Escalation

Privilege escalation in the context of AWS services can take the form of modifying or creating IAM policies and roles, assuming more permissive roles, modifying user profiles, creating temporary credentials, which are not logged as extensively by default.[10] If IAM roles are not scoped appropriately, other AWS services within the affected account (and possibly other accounts) are at risk of compromise.

The extent of the compromise depends on:

- The permissiveness of the IAM role applied to the function
- The type and sensitivity of the event's input data
- The type and sensitivity of any 3rd party data stored in accessible services

4 Logging and Monitoring

Automatic logging and monitoring for AWS Lambda functions is done through CloudWatch, which captures execution start, end, and result unless configured otherwise. Additional logging statements can be added in the code itself, which can also include the plain text contents of event data. CloudWatch can be customized with Rules to respond to specified events, for example a ”failed”

[10]Exploiting AWS Loopholes (Jenko Hwong)

state in an AWS Step Function, and then trigger an action, including another Lambda. Logs can also be collected based on embedded strings with Metric Filtering, which can provide a way to detect disparate events that are related to each other. This can be leveraged to monitor for performance purposes, but also to detect and respond to anomalous activity.

CloudTrail provides audit logs, which contain data on which role or key was used to access a service, and also access activity to the entire AWS account. Along with CloudWatch, it can be used for monitoring and security, particularly for monitoring abnormal KMS key use and behavior, and for tracking API calls. CloudWatch and CloudTrail logs can also be exported for ingestion in a SIEM such as Splunk.

AWS released X-Ray in mid-2017, which is a service specifically designed to monitor distributed applications. It can be integrated (at cost) with a microservices architecture achieved using AWS Lambda, EC2, DynamoDB, and a few others, but not Step Functions at this time. As monitoring is an important part of security, this is a service worth investigating if the application is using covered services.

5 Risk Mitigation

5.1 Identity and Access Management

IAM represents the uppermost boundary of any compromise, and proper configuration is the single best defense against privilege escalation and data exposure. AWS IAM enables the use of roles, and policies attached to use, to grant permissions for services to connect with and affect other AWS services.

```
Policy granting a Lambda full access to a DynamoDB table:
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": "dynamodb:*",
        "Resource": "arn:aws:dynamodb:us-east-2:663910366299:table/Payments"
    }
    ]
}
```

The policy above, which is attached to the role a Lambda can assume, enables that Lambda to perform any action available to the DynamoDB table. The full list of permissions includes actions like DeleteBackup, DeleteTable, RestoreTableFromBackup, TagResource, UpdateGlobalTableSettings, and UpdateItem, just to highlight a few.[11]. In the event that the Lambda given these permissions is compromised, it would be able to do extensive damage.

```
Policy granting a Lambda tightly scoped access to a DynamoDB table:
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "dynamodb:PutItem",
            "dynamodb:UpdateItem"
        ]
        "Resource": "arn:aws:dynamodb:us-east-2:663910366299:table/Payments"
    }
    ]
}
```

The scoped version of this policy allows the Payment Lambda only the access it needs to perform the task it is intended for, and limits the extent of a compromise.

5.2 Restricted Event Data

Given that there is no way of knowing when an external function is compromised, the risk of sensitive data exposure can be mitigated by submitting requests with only the information that the function needs. For applications that are implemented using a Step Function, this functionality is built in and highly configurable depending on the structure of the input data.

Example: A three-Lambda Step Function takes in JSON event data and 1) requests a ship, 2) requests a crew roster, and then 3) processes payment. The following shows what scoped input / output would look like within a step function.

```
Initial Input:

{
    "payment" : {
        "cardnum": "12345678",
        "amount": "1000",
        "first": "c morgan",
        "last": "young"
    },
    "ship" : {
```

[11]DynamoDB API Permissions

```
        "type":"defiant"
    }
}
```

```
Passing all input to final step:

"Pay": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-2:<main_acct>:function:PayForSomething",
    "InputPath": "$",
    "ResultPath": "$.payment.result",
    "OutputPath": "$",
    "End": true
}
```

The InputPath specifies the data to be sent into the identified Resource, while ResultPath, when specified, appends the returned data where indicated. (When not specified, the returned data will override the root of the event data). OutputPath specifies the data to be passed onto the next function.

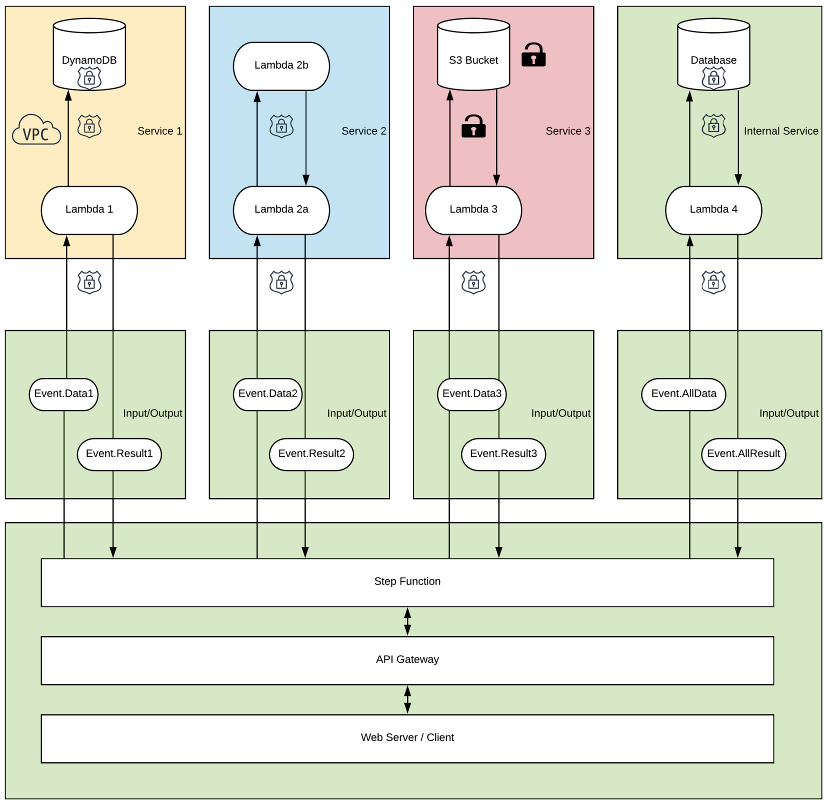
If the event data before this point had additional, sensitive crew roster data attached to it, this would expose that data if the final external Lambda in the sequence was compromised.

```
Pass only scoped input to final step:

"Pay": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-2:<main_acct>:function:PayForSomething",
    "InputPath": "$.payment",
    "ResultPath": "$.payment.result",
    "OutputPath": "$.ship",
    "End": true
}
```

With scoped input and output, the only data exposed to the external Payment Lambda, which is called by a wrapper Lambda within the primary account, is the information needed to complete the payment transaction. The output path also excludes the original payment information from the result returned to the caller.

Example architecture:



5.3 Data Encryption

Function-level data encryption offers another layer of security. Most AWS services allow for both client-side and server-side encryption using its Key Management Service (KMS). KMS keys can also be shared across accounts.

In the following example, the external Payment Lambda processes a payment, then stores the full data to a DynamoDB table. Before doing so, the Lambda encrypts the card number using a shared KMS key, which can be used only by that Lambda and the account that utilized it. In a case where the external Lambda's database is accessible to multiple accounts, this would provide another layer of protection by ensuring that only the two parties can decrypt the plaintext card data. This would be the case even another party had the correct privileges to access the data in that table.

```
def lambda_handler(event, context):
    # filler for processing payment
    process_payment(event["cardnum"])

    card = encrypt_object(event["cardnum"], event["kms_key"])

    response = table.put_item(
        Item={
            'TransactionID' : event["transactionID"],
            'CardNum': card
        }
    )


def encrypt_object(card, kms_key):
    session = boto3.session.Session()
    kms_client = session.client('kms')
    ciphertext = kms_client.encrypt(
        KeyId=kms_key,
        Plaintext=bytes(card, 'utf-8'),
    )
```

In the above code snippets, the ARN for the KMS key is passed in with the event object. The key can only be accessed and used to encrypt and/or decrypt depending on the policies applied to it, which specify the account(s) allowed to perform those actions. In this instance, the permissions assigned are static, and the setup required configuration within both accounts.

```
Key policy snippet: external account access granted to internal KMS key:

{
    "Sid": "Allow use of the key",
    "Effect": "Allow",
    "Principal": {
    "AWS": [
        "arn:aws:iam::<internal_acct>:role/service-role/PayForSomething-role-i5k2k9fu",
        "arn:aws:iam::<external_acct>:root"
        ]
    },
    "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:DescribeKey"
        ],
    "Resource": "<resource_list>"
}
```

6 Recommendations for a Distributed Development Environment

6.1 Runtime and Code Vulnerabilities

Whether developing Lambdas to be used by multiple applications across multiple accounts, or for use within one account, defensive coding practices should be utilized to avoid vulnerabilities. This includes sanitizing input, correctly securing keys, vetting 3rd party modules, and configuring a detailed level of logging to catch and respond to a compromise. Given the ”black box” nature of interacting with 3rd party microservices, every attempt should be made to verify the authenticity and validity of input provided.

Under the Shared Responsibility Model promoted by AWS, the developer does not assume responsibility for hardware and OS-level security, but code, data, and IAM are still within their purview.

Some other recommendations, covered in other resources, are the use of AWS API Gateway instead of allowing 3rd parties to call Lambdas directly, and securing and isolating internal resources within VPCs. Reasons to prefer API Gateway include the ability to throttle API calls by outright rejecting them, integration of custom authorizers to handle OAuth and SAML authentication, and stage variables.

6.2 Data Exposure

3rd party serverless functions should be vetted, whenever possible, to ensure that all sensitive data is stored securely. In a marketplace environment, default encryption for specified operations or data types should, ideally, be the bar for entry.

When appropriate, such as in a data storage resource shared by multiple 3rd parties, an additional encryption layer should be utilized per the example in 5.3, which restricts the data’s encryption key to 2 parties.

Additionally, the principle of least privilege should be utilized in this framework, such that no function receives more data than it needs to complete its task. Should one function be compromised, the exposed data needs to be easily identifiable and scoped tightly to limit the extent of damage.

Example: See ”CloudWatch and Data Flow summary”

6.3 Identity and Access Management

Incorrectly configured IAM roles and policies represent an upper bound on the spread of a compromise. Therefore, serverless functions should always follow the principle of least privilege, and only allow tightly scoped permissions to resources, limited to the minimum requirements to complete the function.

7 Conclusion

While serverless platforms obfuscate the base layer of the operational environment required to run developer code, the shared responsibility model still leaves application code, user data, and identity and access management in the purview of the developer. This shifts the security focus away from a perimeter-based model to one that focuses on scoped permissions, securing the connections and relationships between functions, and implementing the principle of least privilege. In a distributed development architecture, there has to exist trust and dependency on 3rd parties to use best practices, while still mitigating the risk of data exposure in the case of a compromise. This can be accomplished by sharing only necessary data, granting limited permissions, and even added additional layers of security such as shared key encryption for sensitive data. A combination of these practices can be used to vet the security of external services when building a serverless microservice application.

8 Sources

Below is a list of selected sources. Listed AWS Documentation is not a complete list of resources used.

AWS Whitepapers:

Security Overview of AWS Lambda - March 2019
Serverless Architectures with AWS Lambda - November 2017
AWS Key Management Service Cryptographic Details

Selected AWS Documentation:

AWS Service Endpoints - Regions, Endpoints and Protocols
DynamoDB Encryption at Rest

AWS Lambda Execution Context
Accessing AWS Resources from a Lambda Function
DynamoDB API Permissions
AWS Lambda Limits

Presentations:

Hacking Serverless Runtimes (Krug & Jones) - August 2017
Exploiting AWS Loopholes (Hwong) - August 2019
Gone in 60 Milliseconds: Intrusion and Exfiltration in Server-less Architectures (R. Jones) - December 2016

Other Sources:

Lambda Internals: Exploring AWS Lambda
Invoke a Lambda across multiple AWS accounts
Serverless Framework: Lambdas Invoking Lambdas
Serverless Architectures Security Top 10 - 2018
Event Injection: Protecting your Serverless Applications - January 2019
AWS Privilege Escalation Vulnerabilities
Downloading logs from Amazon CloudWatch