Serverless API Security

Ayyaz Akhtar[#1], Brendan Lin[#2], Mohit Pradhan[#3], Evan Sandoval[#4], Jim Klopchic[#5]

[#1]*Department of Cybersecurity, New York University*
New York, USA

[1]aia7143@nyu.edu
[2]bl2279@nyu.edu
[3]mp5775@nyu.edu
[4]es6092@nyu.edu
[5]jfk292@nyu.edu

***Abstract*— Best practice to harden serverless APIs that focus on AWS technologies in Application IAM, Networking, and Observability.**

***Keywords*— Cloud Security, Application IAM, Network Security, Observability, Serverless API Security**

I. Introduction

Serverless cloud technologies offer several advantages, including reduced costs, streamlined resource management, and the ability to scale dynamically based on demand. With these benefits come risks in secure coding, identity and access management, data security, configuration management, and observability. This type of architecture is suitable for consumers that require a quick go-live time, have inconsistent data usage, and are cost-efficient (ex: start-ups) [1]. Neglecting these risks can lead to unauthorized access, information disclosure, compliance violations, and loss of trust for the organization.

II. Scope

For our group project, our team will be using the serverless security workshop as a seed for our research [2]. The output of our research will be a workshop that will go over an example API using API gateway, Lambda, and RDS. The API stack will be implemented with some weak cloud security policies and provide best practices to improve these security measures. Within the AWS Shared Responsibility Model, there exists a breakdown of security responsibility between the customer and AWS [2]. As such, we will be focusing on three key aspects.

- Application IAM
- Network Security
- Observability

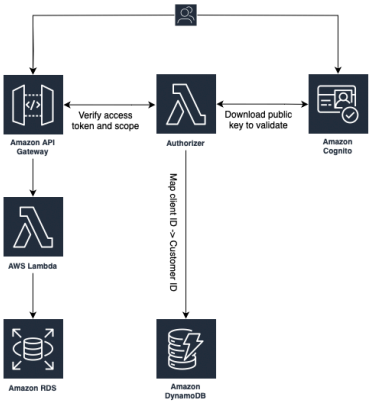
III. Application IAM

1) Gateway

A. Initial Proposal

The initial proposal will include adding Cognito and a Lambda Authorizer. This is intended to provide an authorization mechanism.

Fig. 1 Initial proposal for adding authorization for a serverless API



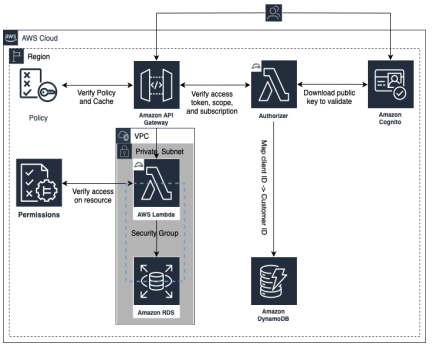
B. Issues encountered

API gateway default setting is set to public and without authorization. This attack vector will need some mitigation strategies. Using an API key is a good starting point, but it has limited scope and is long living. If there are other security request headers, there can be a lot of overhead managing multiple auth headers. For the initial proposal, adding an authorization

Lambda site behind API Gateway, but it lacks permission check on data request context and invocations. In addition, the upstreams are not protected from the gateway, which needs some type of caching and throttling.

C. Final Design

Fig. 2 Final design for adding authorization for a serverless API



The final design will include a Cache policy and verified permissions. These additional features will address some of the issues encouraged with the basic design.

D. Lessons Learned

Lambdas have the ability to restrict access control on managing changes to the source code and deployment. This can be achieved by adding tags that are associated with teams that manage the code.

API keys are not intended for authorization, instead they should be used more for subscription usage and for throttling clients. Instead, having a dedicated lambda authorizer would do the job better. For ease of use, a single lambda authorizer

can be used across multiple accounts and regions.

The Gateway can provide caching on the policies, model API payload to a schema, and set it to private by adding a resource policy. These features are added in the final design to address some of the issues encountered.

E. Future Work

Looking ahead for other opportunities to improve on this design, the next steps would be to integrate with a serverless UI, and IAM access to a serverless RDS instance.

2) Lambda

A. Initial Proposal

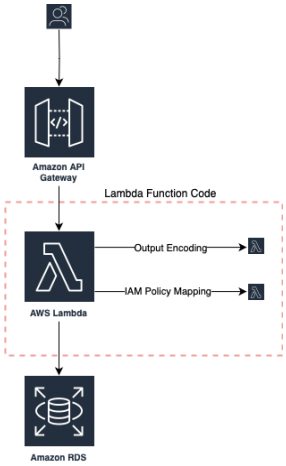


Fig. 3 Proposal design for lambda implementation

Using html encoding libraries within Lambdas will reduce the risk of untrusted user input injecting and executing scripts in user browsers to steal session tokens.

Leveraging function specific IAM Policy Mapping will follow the least privilege principle to prevent functions from interacting with out of scope resources.

B. Issues Encountered

Application session management can be targeted, which poses a major security risk. Additionally, Lambdas should not encode untrusted input to prevent security vulnerabilities. Managing Lambda IAM privileges introduces overhead, requiring careful attention to ensure proper security configurations. Lastly, DOM executable scripts have the capability to share keys, which can further complicate security considerations.

C. Final Design

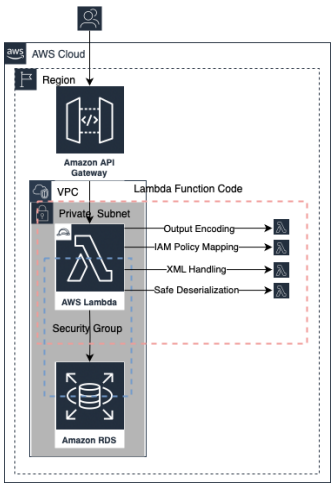


Fig. 4 Final design for lambda implementation

The final design will include XML handling to protect against expansion and external entity based attacks alongside Safe Deserialization to validate untrusted serialized objects

D. Lessons Learned

We put strict controls in place to protect the user of third party libraries, carefully evaluating security features and possible weak points. Additionally, we make effective use of AWS Lambdas functions to handle XML processing and object deserialization. Though powerful, Lambdas might unintentionally expose important code or files if they are not handled appropriately, which is why we continue to be cautious.

E. Future Work

As part of future work we will explore scanning capability available in AWS inspector to perform static code analysis of Lambda. Add testing with fuzzing capability to ensure the robustness of our Lambda function. Incorporating best practices of Secure Software Development Lifecycle.

IV. Network Security

1) VPC

A. Initial Proposal

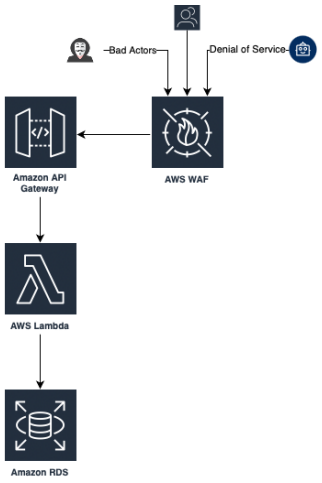


Fig. 5 Proposal design for adding WAF

The initial design was to secure the application using IAM and Cognito along with the AWS WAF. AWS WAF provides the ability to define rich set rules to protect web applications by defining filters to inspect IP addresses, HTTP headers, HTTP bodies, or URI strings from a web request.

B. Issues Encountered

Misconfigured Web application firewall can leave a site vulnerable to some of the attacks like XSS Attacks, SQL Injection attacks. Bad actors can use easily available tools to execute brute force attacks, Denial of Service attacks, etc.

C. Final Design

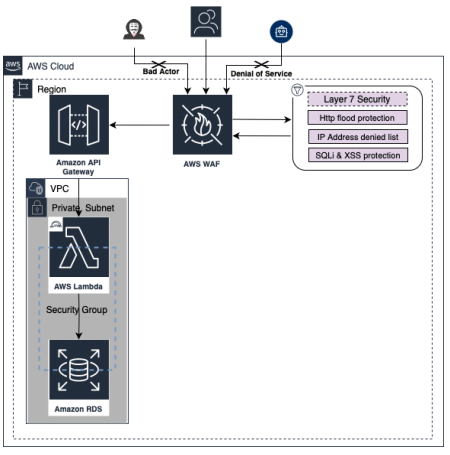


Fig. 6 Final design for adding WAF

In the final design, we configured 3 rules, first a rate limiting rule to protect from flood prevention, SQL injection attack prevention rule and a large body rule to restrict request with large invalid requests.

D. Lessons Learned

AWS WAF has rules to inspect HTTP requests as well as headers to protect against XSS & SQLi attacks. Another basic protection capability is a rate limiting rule which can analyze requests to see if they are originating from the same IP and are flood attacks with the intent of causing denial of service. In addition WAF has a Firewall Account Takeover Prevention (ATP) capability wherein WAF will inspect data using IP and block sending too many unsuccessful request

E. Future Work

The AWS WAF provides a variety of options and rules to protect a web application from some of the common attacks. But these are not sufficient. As part of our future work we will be exploring adding AWS Shield which provides robust protection from DDOS attacks. Add Network firewall to complement the AWS WAF. Explore the 3rd party firewalls to explore the security provided by them.

2) Guard Duty

A. Initial Proposal

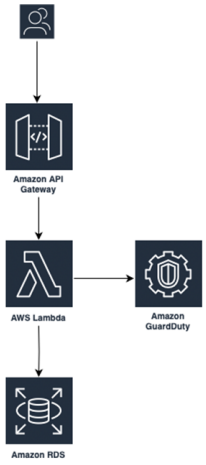


Fig. 7 Initial Proposal design for adding GuardDuty

Running a lambda function that simulates calls to a malicious URL involves executing code within a serverless environment to mimic interactions with potentially harmful web addresses. This allows us to test our system defense against cyberthreats thus enabling us to identify and mitigate vulnerabilities.

B. Issues Encountered

Without logging events or storing data in S3 buckets, and lacking a Security Information and Event Management (SIEM) system for further analysis, this may limit our ability to comprehensively monitor and analyze potential security breaches, leaving our infrastructure vulnerable to undetected threats and attacks.

C. Final Design

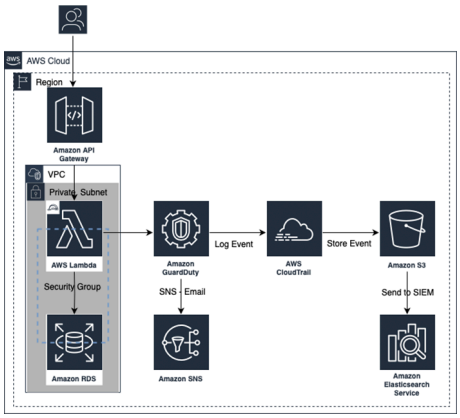


Fig. 8 Proposal design for adding GuardDuty

In our final design, we incorporate Amazon SNS for email alerts, ensuring timely notifications. Additionally, we integrated AWS cloudtrail for comprehensive logging capabilities. The cloudtrail logs are stored in an S3 Bucket and also logs are sent to Amazon Elasticsearch.

D. Future Work

This incident emphasizes how crucial it is to have efficient systems for storage, analysis, and detection in addition to monitoring.

E. Future Work

Implement Amazon SNS for email notifications as well as cloudtrail with an S3 bucket, will also implement Amazon Elasticsearch for further analysis.

V. OBSERVABILITY

1) Cloud Trail

A. Initial Proposal

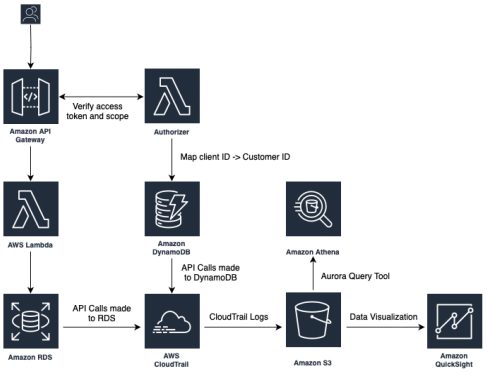


Fig. 9: Initial Proposal for Observability

The initial proposal for the observability portion, as shown in Figure 9, represents the expansion of CloudTrail monitoring for RDS and the usage of Athena and Quicksight as standalone services using the standard architecture. Quicksight is a common business intelligence (BI) tool that supports several data sources including S3, Athena, Aurora, and other unique SQL databases [4].

B. Issues Encountered

The standard architecture monitors API calls that are made to and from DynamoDB, as shown in the progression from the database to CloudTrail, then log storage within S3. The natural progression was to use the logs stored as a data source for Quicksight, however due to its inherently unstructured data format (noSQL), it was unsupported (in contrast to relational databases where data is structured/semi-structured).

Another issue that appeared was the usage of S3 as a data source. There are file size and capacity limitations when using S3- which could be an issue when considering the number of API calls that would likely be made from both databases when queries are made [5]. Using S3 as a primary data source implies that it may store raw or unstructured data, which can lead to incorrect Quicksight visualizations.

Finally, using Athena as a standalone service to query S3 data proved to be impractical. Athena is a service intended to query and analyze data (unstructured/structured) within S3 [6]. However, as mentioned- the queries that Athena produces can

be used as a data source for Quicksight, thus making visualizations more intuitive and accurate.

C. Final Design

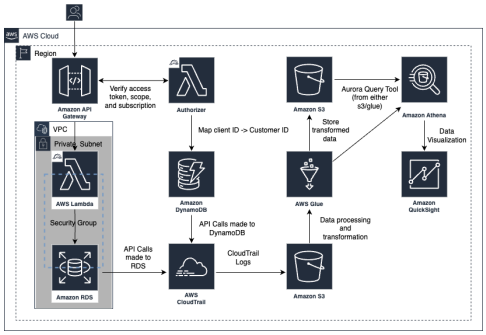


Fig. 10: Final design for Observability

The final design for observability, as shown in Figure 10, represents adjustments made to Figure 9, solving the issues listed in the *Issues Encountered* section (5B). Primarily, the addition of AWS Glue (ETL service) and using Athena as a data source for Quicksight are the implementations that work to drive a serverless analytics feature for the architecture [7][8][9].

D. Lessons Learned

AWS Glue is an ETL (extract, transform, load) service that focuses on "cleaning" data and is applicable to most data sources, including RDS and DynamoDB. Glue is commonly used when there is an analytics use case and data may not be uniform (for example, DynamoDB data). From the "cleaned" Glue data located in a new S3 bucket, Athena can effectively query this data without significant issues [10]. Finally, Athena queries can be utilized as a data source for Quicksight- transforming the standard architecture into a serverless analytics based pipeline.

Additionally, specifically for RDS, there are features known as Performance Insights and Enhanced Monitoring that can be used in conjunction with the analytics pipeline to provide monitoring specific capabilities. Performance Insights focuses on monitoring database load, CPU usage, and other session attributes which can be used to determine scaling for the RDS [11][12]. Enhanced Monitoring provides granular operating system metrics that RDS is run on, including memory, network, process, and disk I/O usage [13]. These are additional monitoring services that can be used specifically with RDS. DynamoDB does not have unique monitoring tools, rather it can be monitored using standard AWS services including CloudWatch, CloudTrail or the DynamoDB dashboard [14].

E. Future Work

With the current architecture, there are services that would integrate well. The first would be AWS SageMaker, a Machine Learning (ML) service that can be used to build predictive models based on the current Quicksight visualization and account for future database API call usage [15]. The second would be to implement Aurora Serverless v2- as it supports the serverless aspect that this architecture represents, with a focus on auto-scaling and provisioning as required [16]. Aurora serverless V1 can also be used, though V2 has several improvements over V1 including multi-AZ scaling based on memory and scaling that can occur at any time [17]. Finally, we can also explore the usage of Zero-ETL integration with Redshift [18]. As its name implies, this method removes the requirement of ETL tools- in this case, AWS Glue. Additionally, Redshift is a fully managed data warehouse service that can load and query data using SQL- also removing the need for Athena. It can be used with Quicksight or other similar 3rd party BI tools to visualize queried data.

References

[1] https://www.cloudflare.com/learning/serverless/why-use-serverless
[2] https://catalog.us-east-1.prod.workshops.aws/workshops/026f84fd-f589-4a59-a4d1-81dc543fcd30/en-US
[3] https://aws.amazon.com/blogs/architecture/architecting-secure-serverless-applications
[4] https://docs.aws.amazon.com/quicksight/latest/user/supported-data-sources.html
[5] https://docs.aws.amazon.com/quicksight/latest/user/data-source-limits.html
[6] https://docs.aws.amazon.com/athena/latest/ug/when-should-i-use-ate.html
[7] https://aws.amazon.com/blogs/big-data/transform-data-and-create-dashboards-simply-using-aws-glue-databrew-and-amazon-quicksight/
[8] https://aws.amazon.com/blogs/big-data/a-serverless-operational-data-lake-for-retail-with-aws-glue-amazon-kinesis-data-streams-amazon-dynamodb-and-amazon-quicksight/
[9] https://aws.amazon.com/blogs/big-data/visualize-amazon-dynamodb-insights-in-amazon-quicksight-using-the-amazon-athena-dynamodb-connector-and-aws-glue/
[10] https://docs.aws.amazon.com/athena/latest/ug/glue-athena.html
[11] https://aws.amazon.com/rds/performance-insights/faqs/
[12] https://docs.aws.amazon.com/prescriptive-guidance/latest/amazon-rds-monitoring-alerting/db-instance-performance-insights.html
[13] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/USER_Monitoring-Available-OS-Metrics.html
[14] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/monitoring-automated-manual.html
[15] https://docs.aws.amazon.com/quicksight/latest/user/sagemaker-canvas-integration.html
[16] https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.html
[17] https://dev.to/aws-builders/aurora-serverless-v1-to-serverless-v2-comparison-migration-and-bluegreen-deployment-4aa8
[18] https://aws.amazon.com/blogs/big-data/announcing-zero-etl-integrations-with-aws-databases-and-amazon-redshift/
[19] https://docs.aws.amazon.com/whitepapers/latest/guidelines-for-implementing-aws-waf/guidelines-for-implementing-aws-waf.html
[20] https://raw.githubusercontent.com/OWASP/Serverless-Top-10-Project/master/OWASP-Top-10-Serverless-Interpretation-en.pdf
[21] https://github.com/OWASP/Serverless-Top-10-Project