AWS Lambda Security
Best Practices
Building and Deploying Secure AWS Lambda Serverless Applications
Table of Contents

3  Preface
4  An Overview of AWS Lambda
5  Serverless Security
7  Serverless Security’s Top Risks
12 Preventing Cloud Lateral Movement with AWS IAM
14  Logging and Audit Trails for AWS Lambda
18  Scalability and How to Avoid DoS Attacks in AWS Lambda
25  Compliance and Governance with AWS Config
29  API Gateway Security
35  Securing Serverless with Prisma Cloud
Preface

This e-book is meant to serve as a security awareness and education guide for organizations developing serverless applications on AWS Lambda.

With many organizations still exploring serverless architecture or just taking their first steps in the serverless world, we believe this information is critical for their success in building robust, secure, and reliable AWS Lambda–based applications.

We hope you adopt and use the best practices outlined herein during the process of designing, developing, and testing AWS Lambda serverless applications to minimize your security risk.
An Overview of AWS Lambda

**AWS Lambda** is an event-driven, serverless computing platform provided as part of Amazon Web Services (AWS®). It is a computing service that runs code in response to events and automatically manages the computing resources that code requires. AWS Lambda lets organizations run code without provisioning or managing servers, and billing applies only when compute time consumed, with no charge when code is not running.

With AWS Lambda, organizations can run code for virtually any type of application or backend service, all with zero administration. The benefits of AWS Lambda are:

- No resource provisioning or server management required to automatically run function code.
- Automatic scaling of serverless applications by running code in response to event triggers. Serverless code runs in parallel, and each process triggers individually, scaling precisely with the size of the workload.
- Billing according to every 100 milliseconds in which code executes and the number of times the code is triggered.
Serverless Security

From a software development perspective, organizations adopting serverless architecture can focus on core product functionality and completely disregard the underlying operating system (OS), application server, or software runtime environment.

By developing applications using serverless architecture, you relieve yourself of the daunting task of constantly applying security patches for the underlying OS and application servers—these tasks become the responsibility of the serverless architecture provider.

With serverless architecture, the provider is responsible for securing the data center, network, servers, OS, and their configurations. However, application logic, code, data, and application-layer configurations still need to be robust and resilient to attacks, and this is the application owners’ responsibility.

Figure 1: Shared responsibility for serverless security
Serverless Security Considerations

Serverless architecture introduces a new set of issues to consider when securing applications:

- **Increased attack surface**: Serverless functions consume data from a wide range of event sources, such as HTTP APIs, message queues, cloud storage, and internet of things (IoT) device communications. This may increase the attack surface, especially when such events use complex message structures, many of which standard application-layer protection, such as web application firewalls (WAFs), cannot inspect.

- **Attack surface complexity**: Since serverless architecture is still a new concept, the attack surface in such architectures can be difficult for some to understand. Many software developers and architects are still inexperienced with the security risks and measures required to secure such applications.

- **Overall system complexity**: Visualizing and monitoring serverless architecture is still more complex than doing so in standard software environments.

- **Inadequate security testing**: Performing security testing for serverless architecture is more complex than testing standard applications, especially when such applications interact with remote, third-party services or backend cloud services, such as NoSQL databases, cloud storage, or stream processing services.

- **Traditional security**: Organizations using serverless architecture do not have access to the physical or virtual server or its OS, so they are not at liberty to deploy traditional security layers, such as endpoint protection, host-based intrusion prevention, or WAFs. In addition, existing detection logic and rules have yet to be “translated” to support serverless environments.
Serverless Security’s Top Risks

Function Event Data Injection
Injection flaws in applications, among the most common risks to date, have been thoroughly covered in many secure coding best practice guides. AWS Lambda functions can consume event input from different event sources, and such input may include different message formats and encoding schemes, depending on the type of event and its source. The different data elements of these messages may contain attacker-controlled or otherwise dangerous inputs.

Broken Authentication
Since serverless architecture promotes a microservices-oriented system design, applications built for such architectures often contain dozens or hundreds of distinct AWS Lambda functions, each with its own specific purpose. These functions are weaved together and orchestrated to form the overall system logic. Some functions may expose public web APIs while others may serve as an internal “glue” between processes. In addition, some functions may consume events of different source types. Applying robust authentication schemes, which provide access control and protection to all relevant functions, event types, and triggers, is a complex undertaking that can easily go awry if not done carefully.

Unsecure Deployment Configuration
Cloud services in general and serverless architecture in particular offer many customization and configuration settings to allow them to adapt to specific needs, tasks, or environments. Some of these settings have critical implications for the overall security posture of your applications, and the vendor’s default settings may not always suit your needs, so it’s important to pay close attention to these. One extremely common security weakness in applications that use cloud-based storage is incorrectly configured cloud storage authentication/authorization.
Over-Privileged IAM Permissions and Roles
You should always follow the principle of least privilege for serverless applications. This means only giving a serverless function the privileges essential to performing its intended logic. Since serverless functions usually follow microservices concepts, many serverless applications contain dozens, hundreds, or even thousands of functions, which can quickly become tedious to manage. In such scenarios, some organizations are forced to use a single identity and access management (IAM) permission model or security role for all functions, essentially granting each function full access to all other components in the system. In a system where all functions share the same set of over-privileged permissions, a vulnerability in a single function can eventually escalate into a system-wide catastrophe.

Inadequate Monitoring and Logging
AWS provides several logging facilities, such as Amazon CloudWatch and AWS CloudTrail®, but in their basic, out-of-the-box configuration, these are not always suitable for providing full security event audit trails. To achieve adequate real-time security event monitoring with proper audit trails, serverless developers and their DevOps teams have to stitch together logging logic that will fit their organizational needs.

Unsecure Third-Party Dependencies
Generally, a serverless function should be a small piece of code that performs a single, discrete task. Often, to perform this task, the serverless function will need to depend on third-party software packages and open source libraries as well as consume third-party remote web services through application programming interface (API) calls.
It is important to note that even the most secure serverless function can become vulnerable when importing code from a vulnerable third-party dependency.

Unsecure Application Secrets Storage
As applications grow in size and complexity, there is a need to store and maintain “application secrets,” such as API keys, database credentials, encryption keys, or sensitive configuration settings. One of the most frequent mistakes in this regard is storing secrets in a plaintext configuration file that is part of the software project. In such cases, any user with “read” permission on the project can access these secrets. Things get much worse if the project is stored in a public repository. Another common mistake is to store these secrets in plaintext as environment variables. While environment variables are a useful way to persist data across serverless function executions, such environment variables can leak and reach the wrong hands in some cases.

Denial of Service and Financial Resource Exhaustion
In the past decade, we have seen a dramatic increase in the frequency and volume of denial-of-service (DoS) attacks. Such attacks are now one of the primary risks facing almost every company exposed to the internet. While serverless architecture brings a promise of automated scalability and high availability, it does impose some limitations and issues that require attention.
Execution Flow Manipulation

Manipulation of application flow may help attackers subvert application logic. Using this technique, an attacker can bypass access controls, elevate user privileges, or even mount a DoS attack. Application flow manipulation is not unique to serverless architecture—it is a common problem in many types of software—but serverless applications are unique in that they often follow the microservices design paradigm and contain many discrete functions, chained together in a specific order, that implement the overall application logic.

In a system with multiple functions, where each function may invoke another function, the order of invocation might be critical to achieving the desired logic. Moreover, the design might assume certain functions are only invoked under specific scenarios and only by authorized invokers.

Improper Exception Handling and Verbose Errors

Existing ways of performing line-by-line debugging of serverless applications are rather limited, and they are more complex than the debugging capabilities available when developing standard applications. This is especially true in cases where a serverless function is using cloud-based services that are not available when debugging the code locally.

This factor forces some developers to adopt the use of verbose error messages and enable debugging environment variables, and then, when moving the code to the production environment, they may forget to clean it first.

Ungoverned Serverless Assets

As more and more applications are developed using serverless architecture, the number of deployed functions and associated cloud resources will proliferate. Without proper governance, organizations could end up with unregistered and ungoverned resources, which may pose security risks if not deleted.
Cross-Execution Data Persistency

AWS Lambda provides developers with local disk storage, which can be used to temporarily persist data during function execution. This data is not automatically purged between executions in the same runtime environment (e.g., container), and if not handled properly, sensitive data may leak across different executions of the same function in that runtime environment.

Leakage of IAM Credentials

When an AWS Lambda function executes, it uses the temporary security credentials received by assuming the IAM role the developer granted to that function. When assuming a role, the assuming entity receives three parameters from AWS Security Token Service (STS):

- AWS_SECRET_ACCESS_KEY
- AWS_ACCESS_KEY_ID
- AWS_SESSION_TOKEN

If an attacker gains access to these three extremely sensitive tokens, he or she can impersonate your function and access any resources available to it. An attacker can even do this remotely, from outside the AWS Lambda environment (such as from a home desktop).
Preventing Cloud Lateral Movement with AWS IAM

The AWS IAM model is one of the most granular and powerful permission models among cloud providers. However, as the saying goes, with great power comes great responsibility.

When you create IAM policies, on AWS and in general, you should always follow the principle of least privilege for permissions. In the case of AWS Lambda functions, the role you assign to a function will dictate the permissions the function will have while it executes. Under certain conditions, this permissions model might save your sensitive data. With an overly permissive IAM role assigned to a function, an attacker may leverage an application-layer vulnerability in your function to perform lateral movement into other resources in your AWS account.

When you design an application for AWS Lambda, you should use the microservices model not only to break things down into more manageable, logical components, but also to essentially compartmentalize different capabilities from one another. Such a design, coupled with a well-configured and strict IAM permissions model, will go a long way in reducing the blast radius when one of the components (functions) is vulnerable and abused. In essence, this is similar to how bulkheads are used in a ship—you are creating watertight compartments that can contain water in the case of a hull breach.

```
1 service: insecure-project-demo
2 provider:
3   name: aws
4   runtime: python3.6
5   region: us-east-1
6   profile: demo
7   iamRoleStatements:
8     - Effect: Allow
9       Action:
10      - s3:*
11       Resource: "*"
12     functions:
13       demo-main:
14         handler: handler.main
15         events:
16           - sns: arn:aws:sns:us-east-1:************:production-topic
17           - serverless-puresec-cli
```

Figure 4: Overly permissive serverless.yml configuration file
Auto-Generate Least-Privileged IAM Roles

To help you save precious time, we released an open source tool for AWS Lambda security. You can also use this as a serverless framework plugin that automatically generates AWS IAM roles with the fewest privileges required by your functions. The tool:

- Saves you time by automatically creating IAM roles for you
- Reduces the attack surface of your AWS Lambda functions
- Helps create least-privileged roles with the minimum required permissions
- Currently supports Node.js and Python runtimes
- Currently supports DynamoDB, Kinesis, KMS, Lambda, S3, SES, SNS, and Step Functions services
- Works with the serverless framework

Developers can add the tool as a part of every deployment process. Installation is simple: `npm install --save-dev serverless-puresec-cli`

You can then start running it by executing:

```
serverless puresec gen-roles --function demo-main
```
Logging and Audit Trails for AWS Lambda

The lifecycle of a cyberattack usually begins with a reconnaissance phase, wherein attackers scout for weaknesses and potential vulnerabilities they may later use to exploit the system. Looking back at major successful cyber breaches, one key element that always gave attackers an advantage was the lack of real-time incident response, which stemmed from failure to detect early signals of the attacks. Had the victim organizations had efficient, real-time security event monitoring and logging, many successful attacks could have been prevented.

A key aspect of serverless architecture is the fact that it resides in a cloud environment, outside of the organizational data center perimeter. As such, on-premises or host-based security controls become irrelevant as viable protection. This, in turn, means that any processes, tools, or procedures developed for security event monitoring and logging become ineffective.

AWS provides two logging tools relevant for keeping an eye on potential security incidents in AWS Lambda: Amazon CloudWatch and AWS CloudTrail.

Amazon CloudWatch

CloudWatch is a monitoring service built to provide application owners with data and actionable insights to monitor applications, respond to performance-related events, optimize resource utilization, and get a unified view of operational application health. CloudWatch collects log data, metrics, and events, providing users with a unified view of AWS cloud resources, applications, and services. In the context of AWS Lambda security, CloudWatch should be used for:

- Monitoring “concurrent executions” account metrics and investigating spikes in AWS Lambda concurrent executions
- Monitoring AWS Lambda throttling metrics
• Monitoring AWS Lambda errors metrics, specifically for timeouts, which may indicate a DoS attack
• AWS Lambda errors related to application failures

AWS CloudTrail

Tracking events in your serverless functions is a start on the path to rock-solid security, but a wealth of activities in any serverless platform can have unexpected effects on your application’s security. AWS CloudTrail is a service designed to enhance your AWS Lambda security and, in general, increase your control over what’s going on in your cloud environment.

CloudTrail is enabled by default on every AWS account when the account is created. When a supported event activity occurs in AWS Lambda, that activity is stored in a CloudTrail event along with other AWS service events in the “Event History” console.

To maintain an ongoing record of events in an AWS account, users must first create a “trail.” This enables CloudTrail to deliver log files to an Amazon S3 bucket. Once stored in S3, logs can be queried using SQL queries on the trails through Amazon Athena. This is far more efficient than manually sifting through JSON log dumps.

By default, when you create a trail in the AWS management console, the trail applies to all AWS regions, logs events from all regions in AWS, and delivers the log files to the Amazon S3 bucket that you specify. You can also configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.
Note that not all AWS Lambda actions are available. At the time of writing, the following AWS Lambda actions are logged in CloudTrail:

GetFunctionConfiguration, GetPolicy ListEventSourceMappings,
ListFunctions, RemovePermission,
UpdateEventSourceMapping, UpdateFunctionCode,
UpdateFunctionConfiguration.

If you turn on data event logging, CloudTrail will also log function invocations so you can see which identities are invoking the functions and how frequently. Each Lambda invocation is logged in CloudTrail as it occurs. The event payload, however, is not logged. So verifying the source caller might be possible, but verifying event structure is not.

Compliance requirements for various government data protection regulations (e.g., GDPR, SOC 2) stipulate that an application must be able to provide logs of application behavior. CloudTrail was designed with this in mind.

Figure 6, taken from AWS documentation, is a sample log value, demonstrating a configuration change to a Lambda function. We can see the details of the user who performed the change and the nature of the change. In this case, the user configured the Lambda function “test-s3-change-dev-hello” to receive notifications from the S3 bucket named “www.some-bucket.xyz.”
CloudTrail can track changes throughout your AWS account, allowing you to trace any infrastructure modification back to its source. This includes details on the login that initiated a configuration change, timestamps, and other associated data that will allow you to fully track your application’s environment configuration.

One of the most significant benefits of enabling CloudTrail for your AWS Lambda serverless functions comes from the built-in automation. You can set up notifications, messages, and alerts that are triggered by configuration events in your AWS ecosystem, enabling you to react to configuration errors and potential security risks as they are introduced. For example, a CloudWatch alert can be automatically triggered when a specific type of activity occurs in an S3 bucket.

Automation is critical in serverless developments as a whole, and that’s no less true for serverless security. With strict use of automated verification and validation, you can test and document your serverless execution environment, creating a robust and predictable application that has all the benefits of a serverless application while enjoying the security of more traditional architecture.
Scale is usually the first thing that usually comes to mind when we think of the word “serverless.” One of the biggest perceived advantages of going serverless is that you don’t need to worry about scale or capacity planning anymore because the cloud provider does all the heavy lifting for you. In reality, though, this is only partially true. When designed correctly, serverless applications are indeed much more resilient to spikes in traffic and can easily scale to support high bandwidth. However, there are certain limitations, and you should follow certain best practices to ensure things go as planned. Otherwise, serverless applications can be just as vulnerable to DoS attacks as any other application out there.
Invocation Types

Lambda functions can be invoked synchronously or asynchronously. Synchronous invocation means that the service or API that invoked the Lambda function will wait for the function to finish running. On the other hand, when a Lambda function is invoked asynchronously, the invoker does not wait for a result.

<table>
<thead>
<tr>
<th>Services</th>
<th>Event source type</th>
<th>Invocation type</th>
<th>When throttled</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Gateway, Cognito, Lex, Alexa</td>
<td>Not stream-based</td>
<td>Synchronous</td>
<td>No retry built in</td>
</tr>
<tr>
<td>CloudFront</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3, CloudWatch, (Logs &amp; Events)</td>
<td>Not stream-based</td>
<td>Asynchronous</td>
<td>Up to 2 retries</td>
</tr>
<tr>
<td>CloudFormation, SNS, SES, Config, CodeCommit</td>
<td>not stream-based</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DynamoDB Streams, Kinesis</td>
<td>Poll-based and stream-based</td>
<td>Synchronous</td>
<td>Automatic retry for the failed batch of records until the data is either expired or successfully processed. The retry mechanism is blocking</td>
</tr>
<tr>
<td>SQS</td>
<td>Poll-based and stream-based</td>
<td>Synchronous</td>
<td>Automatic retry for the failed batch of records until the Visibility Timeout period expires or the batch is successfully processed</td>
</tr>
</tbody>
</table>

When you manually invoke a Lambda function (using either AWS command line interface [CLI] or software development kit [SDK]) you can specify what invocation type you want to use. However, when you use an AWS service as a trigger, the invocation type is predetermined for each service. See Table 1 for a summary of different services, their invocation types and their behavior upon throttling.
Scalability Considerations

Synchronous Invocations

AWS documentation states: “If the function is invoked synchronously and is throttled, Lambda returns a 429 error and the invoking service is responsible for retries. The ThrottledReason error code explains whether you ran into a function level throttle (if specified) or an account level throttle. Each service may have its own retry policy.”

As an example, API Gateway events use synchronous invocations. An attacker who can control the number of requests sent to API Gateway will be able to throttle it, causing a DoS event. Applications that use synchronous invocations are easier for attackers to target since the feedback is immediate and attackers can quickly figure out whether their attacks succeed.

The same idea applies to other event sources in the same category. An attacker can take advantage of pre-authentication Cognito triggers or mount an attack against a chatbot application by causing throttling through the Lex intent integration.

Asynchronous Invocations

AWS documentation states: “If your Lambda function is invoked asynchronously and is throttled, AWS Lambda automatically retries the throttled event for up to six hours, with delays between retries. For example, CloudWatch Logs retries the failed batch up to five times with delays between retries. Remember, asynchronous events are queued before they are used to invoke the Lambda function. You can configure a Dead Letter Queue (DLQ) to investigate why your function was throttled.”

Scalability Considerations

Synchronous Invocations

AWS documentation states: “If the function is invoked synchronously and is throttled, Lambda returns a 429 error and the invoking service is responsible for retries. The ThrottledReason error code explains whether you ran into a function level throttle (if specified) or an account level throttle. Each service may have its own retry policy.”

As an example, API Gateway events use synchronous invocations. An attacker who can control the number of requests sent to API Gateway will be able to throttle it, causing a DoS event. Applications that use synchronous invocations are easier for attackers to target since the feedback is immediate and attackers can quickly figure out whether their attacks succeed.

The same idea applies to other event sources in the same category. An attacker can take advantage of pre-authentication Cognito triggers or mount an attack against a chatbot application by causing throttling through the Lex intent integration.

Asynchronous Invocations

AWS documentation states: “If your Lambda function is invoked asynchronously and is throttled, AWS Lambda automatically retries the throttled event for up to six hours, with delays between retries. For example, CloudWatch Logs retries the failed batch up to five times with delays between retries. Remember, asynchronous events are queued before they are used to invoke the Lambda function. You can configure a Dead Letter Queue (DLQ) to investigate why your function was throttled.”
Let’s take Amazon S3 as an example. An application that lets a user control the frequency with which objects are uploaded to the bucket, and as a result the frequency of concurrent executions of Lambda functions, has the potential to be throttled.

AWS states that Lambda “automatically retries the throttled event for up to six hours,” meaning that a long enough DoS attack can eventually cause data loss.

Another danger with asynchronous invocations, besides the possibility of being throttled, is the unexpected behavior of the application due to the “retry” mechanism. If your Lambda functions are being invoked more than once, and you designed and planned for only one execution, your application flow might break.

**Poll-Based and Stream-Based Invocations**

AWS documentation states: “AWS Lambda polls your stream and invokes your Lambda function. When your Lambda function is throttled, Lambda attempts to process the throttled batch of records until the time the data expires. This time period can be up to seven days for Amazon Kinesis. The throttled request is treated as blocking per shard, and Lambda doesn’t read any new records from the shard until the throttled batch of records either expires or succeeds. If there is more than one shard in the stream, Lambda continues invoking on the non-throttled shards until one gets through.”

The potential victims here are applications with DynamoDB Streams or Kinesis Streams triggers. An attacker can send a malformed batch of events (meaning events that will trigger an error during the function’s execution) to the stream, causing the retry mechanism to step up. If not handled properly, this will cause a DoS event since the record processing is blocked.

**Poll-Based and Non-Stream-Based Invocations**

AWS documentation states: “AWS Lambda polls your queue and invokes your Lambda function. When your Lambda function is throttled, Lambda attempts to process the throttled batch of records until it is successfully invoked (in which case the message is automatically deleted from the queue) or until the MessageRetentionPeriod set for the queue expires.”
According to AWS, the messages in an Amazon Simple Queue Service (SQS) queue are processed in the following manner: AWS Lambda automatically scales up polling activity until the number of concurrent function executions reaches 1,000, the account concurrency limit, or the optional function concurrency limit, whichever is lowest. SQS supports an initial burst of five concurrent function invocations and increases by 60 concurrent invocations per minute.

Let’s assume we have a Lambda function that takes five minutes to process an event in the queue, and the concurrency limit is 1,000 executions. Since the default MessageRetentionPeriod for a message in the queue is four days, an attacker attempting to cause data loss will have to issue at least a certain number of requests at once—assuming a DLQ is not configured—to succeed. Let’s plug these figures into an equation:

\[ \text{#Requests} = \text{ConcurrentLimit} \times \left( \frac{60}{\text{ProcessingTime}} \right) \times 24 \times 4 \]

In our example, the number of requests required to cause data loss would be 1,152,000. That’s definitely not an easy task.

**Mitigation and Best Practices**

Service-level mitigation:

- API Gateway provides the ability to set quota and throttling criteria.
- For relevant APIs, consider enabling API response caching to reduce the number of calls made to your API endpoint and improve the latency of requests to your API.
- For S3 specifically, consider using SQS as a broker to your Lambda function. By defining a queue instead of Lambda as a destination, you gain the ability to process multiple events at once (and you can define the batch processing size).
• Make sure your code doesn’t “hang” when faced with unexpected input. You should carefully test all edge cases and think of inputs that might cause function timeouts (e.g., regular expression DoS attacks or long payloads). An attacker might be able to exploit such an application-layer weakness.
• The serverless security capabilities of Prisma Cloud provide behavioral runtime protection against a wide range of attacks and can reduce the risk of application-layer DoS as well as unauthorized malicious behavior. The platform also provides unparalleled forensic-level visibility.

Architectural design considerations:
• Design for retry. Always build your Lambda functions in a way that accounts for the possibility of processing the same event more than once.
• Reduce the blast radius by defining a reserved capacity limit to specific Lambda functions so that an attacker can’t use them to consume the entire account capacity.
• Set up a DLQ for Lambda functions with asynchronous event triggers (in SQS integrations, for the queue itself). After retrying the event twice, Lambda will forward it to the DLQ destination (SQS queue or Simple Notification Service [SNS] topic) for further investigation.

Monitoring:
• Monitor your concurrent executions account metrics. You can find more information on how to investigate spikes in AWS Lambda concurrency [here](#).
• Monitor your Lambda throttling metrics.
• Monitor your Lambda errors metric, specifically for timeouts.
• It is highly recommended to set up monitoring and alerts on your AWS charges and billing.
Compliance and Governance with AWS Config

AWS Config Overview
You can generally divide AWS services into two classes: operative services, such as Lambda and S3; and helper/utility services, such as CloudWatch and CloudTrail.

AWS Config belongs to the latter and is among the most useful AWS services, able to dramatically improve your ability to govern your serverless applications. Governance and visibility are some of the most critical components of any effective security strategy.

With AWS Config, you can automatically record and track changes to the configuration of your AWS Lambda based applications and many other AWS services. Whether you need to comply with industry regulations or abide by custom corporate policies, AWS Config rules enable you to implement security policies as code.

AWS Config offers multiple benefits for AWS Lambda security, including:

- **Continuous monitoring**: Monitor and record configuration or code changes for your AWS Lambda functions.
- **Continuous assessment**: Audit and assess the overall compliance of your AWS Lambda function configurations with your organization’s policies and guidelines.
- **Change management**: Track relationships between functions and resources.

Since each organization has its own set of security policies and regulations, AWS lets you create custom rules associated with an AWS Lambda function. The Lambda function will contain the logic that evaluates whether your AWS resources (such as Lambda functions) comply with your rules.
Custom rules can be invoked in response to configuration changes or periodically (e.g., every 24 hours) to review resources previously deployed to the account.

4 Config Rules to Get You Started

Here, we will help you to create four simple AWS Config rules that will boost your AWS Lambda security posture. As a bonus, we already implemented the rules and uploaded them as a project on GitHub®, including an AWS Serverless Application Model (SAM) application, to help you deploy them quickly.

Deployment Instructions (also available on GitHub)

Step 1: Install the dependencies:

```bash
npm install --prefix src/
```

Step 2: Create a deployment bucket:

```bash
aws s3 mb s3://DEPLOYMENT_BUCKET_NAME
```

Step 3: Package your application:

```bash
aws cloudformation package \
  --template-file template.yaml \
  --output-template-file template-packaged.yaml \
  --s3-bucket DEPLOYMENT_BUCKET_NAME
```
Step 4: Deploy the config rules:

```bash
aws cloudformation deploy \
  --template-file template-packaged.yaml \
  --stack-name lambda-config-rules \
  --capabilities CAPABILITY_IAM \n  `# optionally add FunctionShield token` \n  --parameter-overrides FunctionShieldToken=FTOKEN
```

**Rule 1: Detect AWS Lambda Functions That Are Created Directly Through the AWS Web Console**

The AWS Lambda web console saves time when you need to experiment. We’ve all done that when we need to test or prototype something quickly. That said, it’s not the proper way to deploy serverless applications. Before you know it, you end up with numerous functions deployed but no ability to restore older versions or understand why code changed. If your organization follows continuous integration/continuous deployment (CI/CD) best practices, you shouldn’t deploy new functions through the web console, and this rule will keep track of this.

**Rule 2: Detect Multiple AWS Lambda Functions That Are Using the Same Single IAM Role**

You should always try to ensure your Lambda functions don’t share the same AWS IAM execution role to comply with the principle of least privilege. In general, this means there should always be a one-to-one relationship between your AWS Lambda functions and their IAM roles.
Rule 3: Detect AWS Lambda Functions with Multiple Different Event Triggers Configured
When prototyping or experimenting with Lambda functions, you will sometimes enable invocation through multiple cloud native event trigger types. However, if your function will eventually get triggered by a single event type, you should remove inessential triggers, as they increase your attack surface. If your design allows a function to get invoked by multiple triggers, you should reconsider this design, as it violates the “Single Responsibility” principle.

Rule 4: Detect AWS Lambda Functions with Wildcard (*) IAM Permissions
When you create IAM policies (on AWS and in general), you should always follow the principle of least privilege, granting only the permissions required to perform a task successfully. In the case of AWS Lambda functions, the role you assign to a function will dictate the permissions it will have while it executes. Under certain conditions, this permissions model might save your sensitive data. With an overly permissive IAM role assigned to a function, an attacker may leverage an application-layer vulnerability in your function to perform lateral movement into other resources in your AWS account.
Securing APIs with Amazon API Gateway

Overview
API Gateway enables developers to create, publish, maintain, monitor, and secure APIs. Together with AWS Lambda, Amazon API Gateway forms the app-facing part of the AWS serverless infrastructure. With this, you can run a fully managed REST API that integrates with AWS Lambda functions to execute business logic. Amazon API Gateway provides benefits such as:

- Traffic management
- Authorization and authentication
- Monitoring and logging
- API versioning

API Security
Within your system, you are likely to have APIs with different levels of access. The different types of APIs include:

- **Public/Unauthenticated APIs**—APIs that can be consumed by any user, such as endpoints for returning a landing page that is publicly accessible and does not require users to be authenticated.
• **Authenticated user APIs**—APIs that require users to be authenticated, which typically involve updating system state on the user’s behalf (e.g., an API used to submit a vote in an online voting system).

• **Internal APIs**—APIs meant to be used by the system itself, not consumed by client applications directly in the microservices architecture. Such APIs are often used to encapsulate and manage a set of shared resources (e.g., shared resource data that many parts of the system need to access in a microservices architecture). Internal APIs are often very powerful and can be used to update or even delete system state, so they must be well protected.

**Using Amazon API Gateway**

Typically (in a non-serverless application) you would restrict access to internal APIs by using a combination of private VPCs coupled with authorization. However, when you use an API gateway, you lose the ability to create network boundaries with private VPCs. That said, Amazon API Gateway provides efficient access control mechanisms that are implemented at the API gateway level. One of the ways to secure APIs with an API gateway is to use API keys.

**Controlling Access to APIs**

There are multiple controls in place that you can use to control access to APIs. The most common ones are:
- Usage plans using API keys granted to users with usage quota limiting
- AWS IAM roles and policies
- Amazon Cognito user pools
- Lambda authorizer functions for controlling access to API methods using token authentication as well as header data, URL paths, query string parameters, stage variables, or context variables request parameters

API Keys
API keys are string tokens you provide to client application developers to grant access to your APIs. You can use API keys, together with usage plans or Lambda authorizers, to control access to your APIs. Amazon API Gateway can generate API keys on your behalf, or you can import them from a CSV file.

Find out more about how to set up API keys for use with Amazon API Gateway here.

Usage Plans
A usage plan declares who can access one or more deployed APIs (including stages and methods) and defines the volume/frequency at which they can access those APIs. The plan uses API keys to identify clients and measures access to the associated API stages for each key.

Figure 7: Defining a usage plan with Amazon API Gateway
For example, you might allow prospective customers to try your API during a trial period. In that case, you would limit them to a maximum of one request per second, with a request burst of up to five requests per second. At the same time, you would limit such customers to a total of 10 requests per day.

Find out more about how to set up usage plans [here](#).

**Controlling Access to APIs with AWS IAM**

Should you need to grant employees access to internal APIs, you would probably use AWS IAM authentication, rather than API keys.

To allow an API caller to invoke an API, you must first create an IAM policy that permits a specified API caller to invoke the API method for which the IAM user authentication is enabled. You can set this by configuring the method’s “authorizationType” property to `AWS_IAM`, which will require the caller to submit the IAM user’s access keys to be authenticated. After setting the configuration to use AWS IAM authentication, you will need to attach the policy to an IAM user representing the API caller, to an IAM group containing the user, or to an IAM role assumed by the user.

**Amazon Cognito**

Many developers make the mistake of implementing their own authentication systems. Such systems are a good example of “heavy lifting” work that is neither core to your business nor easy to implement correctly. In fact, many data breaches in the past resulted directly from custom-built authentication systems that were poorly designed or implemented. It is highly recommended to always use an authentication service, such as Amazon Cognito, Auth0, or similar. Such services remove the burden of implementing robust authentication systems, allowing you to focus your development efforts and energy on your business logic.
Amazon Cognito provides authentication, authorization, and user management for your applications. End users can sign in with a username and password or by using a third-party web service such as Google, Facebook, or Amazon.

**Amazon Cognito User Pools**

Cognito user pools is a managed identity service that manages everything related to user sign-up and sign-in. It implements all common user management flows out of the box, as well as a host of leading best practices, including multi-factor authentication (MFA) and server-side data encryption. The service stores passwords using the Secure Remote Password (SRP) Protocol so that passwords never need to travel over the wire and are therefore resistant to several relevant attack vectors. You can allow users to access your APIs through Amazon API Gateway by authenticating them with Amazon Cognito User Pools. In such a configuration, Amazon API Gateway will validate the tokens from a successful user pool authentication and use them to grant your users access to resources, including Lambda functions and your APIs.
AWS Lambda Custom Authorizers

A Lambda authorizer is a serverless function that you create to authorize access to your APIs. A Lambda authorizer uses bearer token authentication, such as SAML or OAuth. It can also use information described by HTTP headers, URL path, query string parameters, and so forth. When an API is called by a client and Amazon API Gateway is configured to use a Lambda authorizer, it will invoke the relevant Lambda function. During the invocation, Amazon API Gateway will pass the authorization token that is extracted from a specified request header for the token-based authorizer, or it will pass the incoming request parameters as the input to the authorizer function.
Securing AWS Lambda with Prisma Cloud

Prisma™ Cloud is an end-to-end cloud security suite from Palo Alto Networks. Prisma Cloud provides comprehensive threat detection, governance, and visibility tools as well as real-time workload protection and threat prevention. The serverless security capabilities of Prisma Cloud are based on the acquisition of PureSec, the market-leading serverless security platform.

Prisma Cloud provides extensive security for every cloud layer and workload type, including VMs, containers, serverless, and advanced cloud services.

Serverless App Firewall and Runtime Protection

Prisma Cloud provides automatic defense against application-layer attacks such as SQL injections, remote code execution, attempts to subvert function logic, and unauthorized malicious actions. Protection is initiated when a function is invoked, where the serverless application firewall employs rigorous security algorithms to detect event-data injection attacks. Once event data is found to be legitimate, the function is allowed to run—then, Prisma Cloud’s machine learning-based behavioral protection engine closely monitors function execution to detect and block unauthorized function interactions and operations in real time.
Serverless Security Posture

Prisma Cloud seamlessly integrates with your CI/CD process. During development and build time, serverless projects are statically scanned to pinpoint risks related to overly permissive IAM roles and known vulnerable third-party dependencies. With Prisma Cloud integrated into your CI/CD pipeline, you’ll have the tools you need to ship robust serverless code at all times.

Serverless Visibility

Prisma Cloud integrates deep into your functions, providing unparalleled visibility into application-layer attacks to let you see what your functions are doing in a way you never have before. For each security event, Prisma Cloud customers receive access to forensic data, allowing deep investigations into security incidents, in real time. Prisma Cloud provides simple integrations with existing security information and event management (SIEM) offerings so your Developer, Security, and Operations teams can receive event information and notifications in the tools of their choice.

Start your 30-day free trial of Prisma Cloud
Prisma Cloud is a comprehensive cloud native security platform with the industry’s broadest security and compliance coverage—for applications, data, and the entire cloud native technology stack—throughout the development lifecycle and across multi- and hybrid cloud environments. Prisma Cloud takes an integrated approach that enables security operations and DevOps teams to stay agile, collaborate effectively, and accelerate cloud native application development and deployment. It eliminates the security constraints around cloud native architectures rather than masking them and breaks the security operational silos across the entire application lifecycle, allowing DevSecOps adoption and enhanced responsiveness to the changing security needs of cloud native architectures.