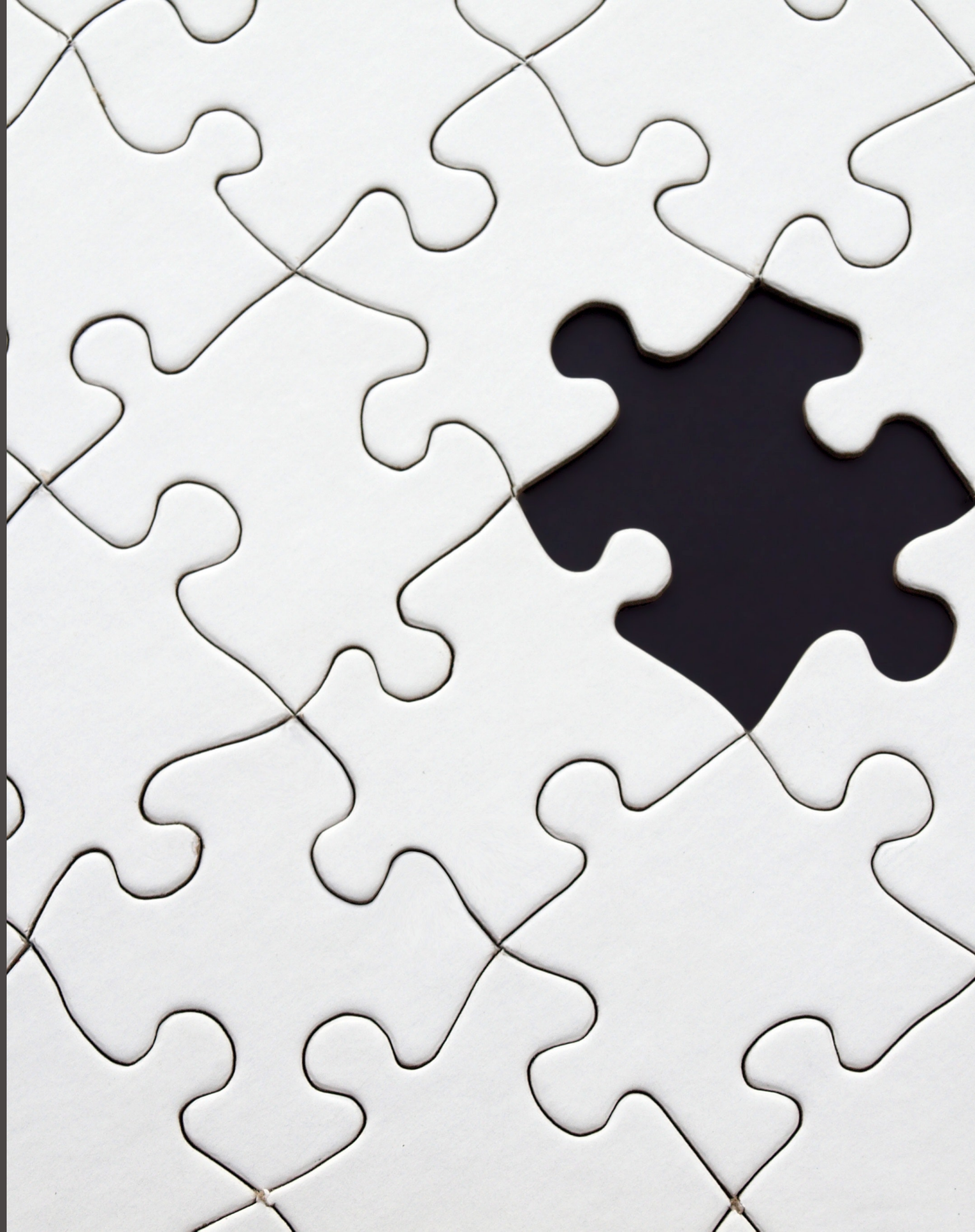
PURESEC

AWS LAMBDA SECURITY BEST PRACTICES

Building & Deploying Secure AWS Lambda Serverless Applications

An eBook by PureSec



What's in this eBook?

TOPICS

- AWS Lambda overview
- Serverless security
- Serverless security: Top Risks
- Identity & access management (IAM)
- Logging & audit trails for AWS Lambda
- Scalability, and how to avoid Denial-of-Service in AWS Lambda
- Compliance and governance for AWS Lambda with AWS Config
- API Gateway Security
- PureSec Serverless Security Platform

PREFACE

This AWS Security Best Practices eBook is meant to serve as a security awareness and education guide for organizations developing serverless applications on AWS Lambda.

As many organizations are still exploring serverless architectures, or just making their first steps in the serverless world, we believe that this eBook is critical for their success in building robust, secure and reliable AWS Lambda based applications.

We hope you adopt this eBook and use it during the process of designing, developing and testing AWS Lambda serverless applications in order to minimize security risks.

An Overview of AWS Lambda

AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of the Amazon Web Services (AWS). It is a computing service that runs code in response to events and automatically manages the computing resources required by that code.

AWS Lambda lets organizations run code without provisioning or managing servers. Billing is done only for the compute time consumed – there is no charge when code is not running.

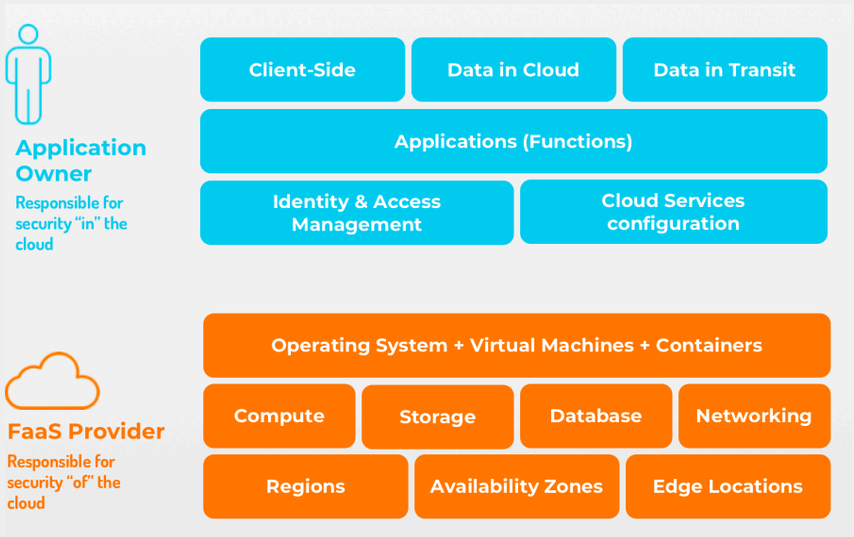
With AWS Lambda, organizations can run code for virtually any type of application or backend service – all with zero administration. The benefits of using AWS Lambda are:

- AWS Lambda automatically runs function code without requiring any resource provisioning or servers management.
- AWS Lambda automatically scales serverless application by running code in response to event triggers. Serverless code runs in parallel and processes each trigger individually, scaling precisely with the size of the workload
- When using AWS Lambda, organizations are charged for every 100ms in which code executes and the number of times the code is triggered.

Serverless Security

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

By developing applications using serverless architectures, you relieve yourself from the daunting task of constantly applying security patches for the underlying operating system and application servers – these tasks are now the responsibility of the serverless architecture provider.



In serverless architectures, the serverless provider is responsible for securing the data center, network, servers, operating systems and their configurations. However, application logic, code, data and application-layer configurations still need to be robust and resilient to attacks, which is the responsibility of application owners.

SERVERLESS SECURITY CONSIDERATIONS

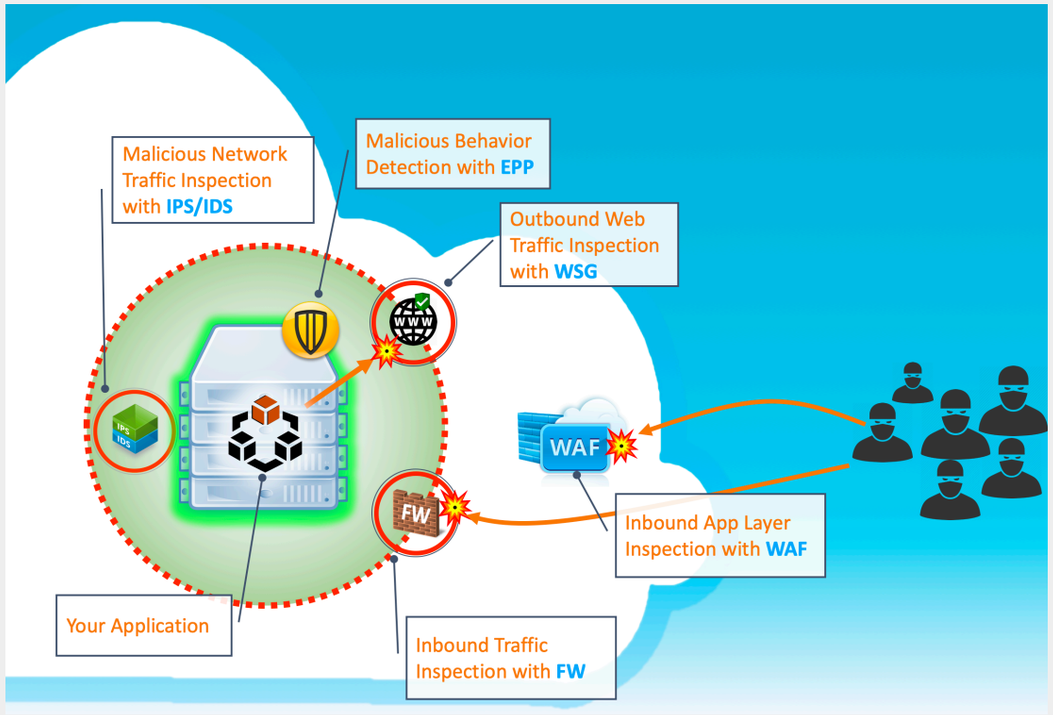
Serverless architectures introduce a new set of issues that must be taken into consideration when securing such applications:

1. Increased attack surface: serverless functions consume data from a wide range of event sources such as HTTP APIs, message queues, cloud storage, IoT device communications and so forth. This may increase the attack surface, especially when such events use complex message structures - many of which cannot be inspected by standard application layer protections such as Web application firewalls.

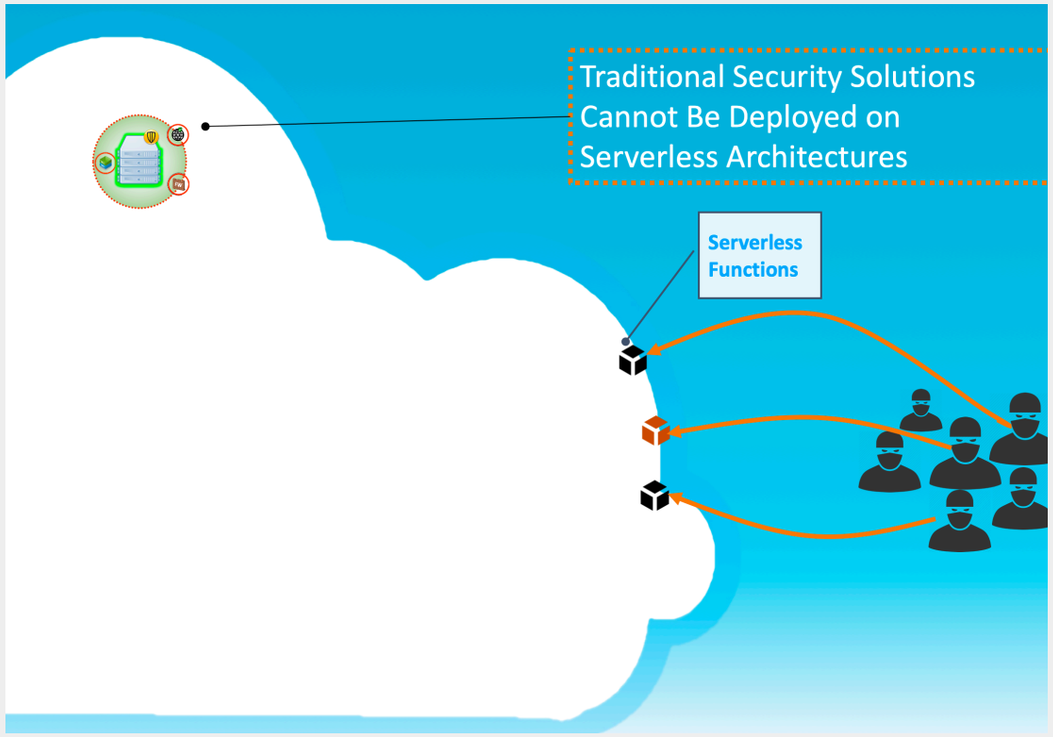
2. Attack surface complexity: the attack surface in serverless architectures can be difficult for some to understand given that such architectures are still rather new. Many software developers and architects have yet to gain enough experience with the security risks and appropriate security protections required to secure such applications Overall system complexity: visualizing and monitoring serverless architectures is still more complex than standard software environments.

3. Inadequate security testing: performing security testing for serverless architectures is more complex than testing standard applications, especially when such applications interact with remote 3rd party services or with back-end cloud services such as NoSQL databases, cloud storage, or stream processing services.

4. Traditional security protections (Firewall, WAF, IPS/IDS): since organizations that use serverless architectures do not have access to the physical (or virtual) server or its operating system, they are not at liberty to deploy traditional security layers such as endpoint protection, host-based intrusion prevention, web application firewalls and so forth. In addition, existing detection logic and rules have yet to be "translated" to support serverless environments.



Traditional applications can be secured with a wide range of solutions



Those solutions can't be deployed on serverless functions

Serverless security
TOP RISKS

- Function Event Data Injection
- Broken Authentication
- Insecure Deployment Configuration
- Over-Privileged IAM Permissions & Roles
- Inadequate Monitoring and Logging
- Insecure 3rd Party Dependencies
- Insecure Application Secrets Storage
- Denial of Service & Financial Resource Exhaustion
- Execution Flow Manipulation
- Improper Exception Handling and Verbose Errors
- Ungoverned serverless assets
- Cross-Execution Data Persistency
- Leakage of IAM credentials
- Out-of-band code deployment

FUNCTION EVENT DATA INJECTION

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

BROKEN AUTHENTICATION

Since serverless architectures promote a microservices-oriented system design, applications built for such architectures may oftentimes contain dozens or even 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 some sort of 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, which may easily go awry if not done carefully.

INSECURE DEPLOYMENT CONFIGURATION

Cloud services in general, and serverless architectures in particular offer many customizations and configuration settings in order to adapt them for each specific need, task or surrounding environment. Some of these configuration settings have critical implications on the overall security posture of the application and should be given attention. The default settings provided by serverless architecture vendors might not always be suitable for your needs. One extremely common weakness that affects many applications that use cloud-based storage is incorrectly configured cloud storage authentication/authorization.

OVER-PRIVILEGED IAM PERMISSIONS & ROLES

Serverless applications should always follow the principle of "least privilege". This means that a serverless function should be given only those privileges, which are essential in order to perform its intended logic. Since serverless functions usually follow microservices concepts, many serverless applications contain dozens, hundreds or even thousands of functions. This in turn means that managing function permissions and roles quickly becomes a tedious task. In such scenarios, some organizations might find themselves forced to use a single permission model or security role for all functions, essentially granting each of them full access to all other components in the system.

INADEQUATE MONITORING & LOGGING

AWS provides several logging facilities such as CloudWatch and CloudTrail. These logs in their basic/out-of-the-box configuration, are not always suitable for the purpose of providing a full security event audit trail. In order to achieve adequate real-time security event monitoring with proper audit trail, serverless developers and their DevOps teams are required to stitch together logging logic that will fit their organizational needs.

INSECURE 3RD. PARTY DEPENDENCIES

In the general case, a serverless function should be a small piece of code that performs a single discrete task. Oftentimes, in order to perform this task, the serverless function will be required to depend on third party software packages, open source libraries and even consume 3rd party remote web services through API calls.

It is important to note that even the most secure serverless function can become vulnerable when importing code from a vulnerable 3rd party dependency.

INSECURE 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 frequently recurring mistakes related to application secrets storage, is to simply store these secrets in a plain text configuration file, which is a part of the software project. In such cases, any user with "read" permissions on the project can get access to these secrets. The situation gets much worse, if the project is stored in a public repository. Another common mistake is to store these secrets in plain text, as environment variables. While environment variables are a useful way to persist data across serverless function executions, in some cases, such environment variables can leak and reach the wrong hands.

DENIAL OF SERVICE / FINANCIAL RESOURCE EXHAUSTION

During the past decade, we have seen a dramatic increase in the frequency and volume of Denial of Service (DoS) attacks. Such attacks became one of the primary risks facing almost every company exposed to the Internet. While serverless architectures bring a promise of automated scalability and high availability, they do impose some limitations and issues which require attention.

EXECUTION FLOW MANIPULATION

Manipulation of application flow may help attackers to subvert application logic. Using this technique, an attacker may sometimes bypass access controls, elevate user privileges or even mount a Denial of Service attack. Application flow manipulation is not unique for serverless architectures – it is a common problem in many types of software. However, serverless applications are unique, as they oftentimes follow the microservices design paradigm and contain many discrete functions, chained together in a specific order which implements the overall application logic.

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

IMPROPER EXCEPTION HANDLING & VERBOSE ERRORS

Existing solutions for performing line-by-line debugging of serverless based applications is rather limited and more complex compared to the debugging capabilities that are available when developing standard applications. This is especially true in cases where the 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, enable debugging environment variables and eventually forget to clean the code when moving it to the production environment.

UNGOVERNED SERVERLESS ASSETS

As more and more applications get developed using serverless architectures, the number of deployed functions and associated cloud resources will proliferate. Without proper governance, organizations might 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 may leak sensitive data across different executions of the same function in the same 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 to that function. When assuming a role, the assuming entity receives 3 parameters from AWS STS (security token service): AWS_SECRET_ACCESS_KEY, AWS_ACCESS_KEY_ID, AWS_SESSION_TOKEN. If an attacker gains access to these 3 extremely sensitive tokens, he/she can impersonate your function and access any resources available to it. Impersonation can even be done remotely from outside the AWS Lambda environment (e.g. the attackers desktop).

OUT-OF-BAND CODE DEPLOYMENT

Serverless applications should always follow the principle of "least privilege". This means that a serverless function should be given only those privileges, which are essential in order to perform its intended logic. Since serverless functions usually follow microservices concepts, many serverless applications contain dozens, hundreds or even thousands of functions. This in turn means that managing function permissions and roles quickly becomes a tedious task. In such scenarios, some organizations might find themselves forced to use a single permission model or security role for all functions, essentially granting each of them full access to all other components in the system.

Preventing Cloud Lateral Movement With

AWS IAM

- AWS IAM overview
- Least-Privilege principle
- Auto-Generating least-privileged roles with PureSec CLI & Serverless framework plugin

The AWS IAM model is one of the most granular and powerful permission models you will find 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 standard security concept of granting the least required privileges - i.e. 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 the function will have while it executes. Under certain conditions, this permissions model, might be the thing that will save your sensitive data. With an over-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, harness the microservices model not only for breaking down things to more manageable logical components, but also because you are essentially compartmentalizing 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 very 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.

AUTO-GENERATE LEAST PRIVILEGED IAM ROLES

In order help you save previous time, PureSec released an open source tool for AWS Lambda security, which you can also use as a Serverless framework plugin, that automatically generates AWS IAM roles with the least privileges required by your functions. Available Features:

- Saves you time - magically creates IAM roles for you
- Reduces the attack surface of your AWS LAMBDA functions
- Helps create least privileged roles with the minimum required permissions
- Currently supported runtimes: Node.js, Python
- Currently supported services: DynamoDB, Kinesis, KMS, Lambda, S3, SES, SNS & Step Functions
- 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
```

```yaml
service: insecure-project-demo

provider:
  name: aws
  runtime: python3.6
  region: us-east-1
  profile: demo
  iamRoleStatements:
  - Effect: Allow
    Action:
      - s3:*
    Resource: "*"
functions:
  demo-main:
    handler: handler.main
    events:
      - sns: arn:aws:sns:us-east-1:***************:production-topic
plugins:
  - serverless-puresec-cli
```

Over-permissive serverless.yml configuration file

```yaml
Resources:
  PureSecDemo-MainRole:
    Properties:
      AssumeRolePolicyDocument:
        Statement:
        - Action: sts:AssumeRole
          Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
        Version: '2012-10-17'
      Path: /
      Policies:
      - PolicyDocument:
          Statement:
          - Action:
            - s3:PutObject
            Effect: Allow
            Resource: arn:aws:s3:::upload-to-s3-and-postprocess/*
          - Action:
            - logs:CreateLogStream
            - logs:PutLogEvents
            - logs:CreateLogGroup
            Effect: Allow
            Resource: arn:aws:logs:us-east-1:**************:log-group:/aws/lambda/insecure-project-demo-dev-demo-main:*
          Version: '2012-10-17'
        PolicyName: PureSecGeneratedRoles
      RoleName: insecure-project-demo-puresec-demo-main
    Type: AWS::IAM::Role
```

Least-privileged IAM security policy

Logging & audit trails for AWS Lambda

LOGGING

- Why log?
- AWS CloudWatch
- AWS CloudTrail

Every cyber “intrusion kill chain” usually commences with a reconnaissance phase – this is the point in time in which attackers scout the application for weaknesses and potential vulnerabilities, which may later be used to exploit the system. Looking back at major successful cyber breaches, one key element that was always an advantage for the attackers, was the lack of real-time incident response, which was caused by failure to detect early signals of an attack. Many successful attacks could have been prevented if victim organizations had efficient and adequate real-time security event monitoring and logging.

One of the key aspects of serverless architectures is the fact that they reside in a cloud environment, outside of the organizational data center perimeter. As such, “on premise” or host-based security controls become irrelevant as a viable protection solution. This in turn, means that any processes, tools and procedures developed for security event monitoring and logging, becomes inapt.

AWS provides two logging facilities, which are relevant for keeping an eye on potential security incidents in AWS Lambda: CloudWatch and CloudTrail.

AWS CLOUDWATCH

CloudWatch is a monitoring service built to provide application owners with data and actionable insights to monitor your applications

and 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 single unified view of AWS cloud resources, applications and services. In the context of AWS Lambda security, CloudWatch should be used for the following:

1. Monitoring 'Concurrent Executions' account metrics, and investigating spikes in AWS Lambda concurrent executions
2. Monitor AWS Lambda throttling metrics.
3. Monitor AWS Lambda errors metric, specifically for timeouts, which may indicate a DoS attack
4. 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 there are a wealth of activities in any serverless platform that can have an unexpected effect on your application's security. AWS CloudTrail is a service designed to help enhance your AWS Lambda security, and in general increase your control over what's going on in your cloud environment.

The CloudTrail service is enabled by default on every AWS account once 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.

In order to maintain an ongoing record of events in an AWS account, users must first create a "trail". A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. Once logs are stored in S3, they can be queried using SQL queries on the trails through AWS Athena. This is by 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. It 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: `AddPermission,CreateEventSourceMapping, CreateFunction, DeleteEventSourceMapping, DeleteFunction, GetEventSourceMapping, GetFunction,`

`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 the frequency of their invocations. 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 (GDPR, SOC2, etc.) stipulate that an application must be able to provide logs of application behavior. And AWS CloudTrail was designed with this case in mind.

On the right is a sample log value, demonstrating a configuration change done on a Lambda function (taken from AWS documentation). We can see the details of the user who performed the change, as well as 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 also track changes throughout your AWS account, allowing you to trace any infrastructure modification


```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDAJ******JPU7I2",
        "arn": "arn:aws:iam::61*******47:user/someuser",
        "accountId": "61*******47",
        "accessKeyId": "ASIAY7******XV7FOV",
        "userName": "someuser",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-09-25T06:57:35Z"
            }
        },
        "invokedBy": "signin.amazonaws.com"
    },
    "eventTime": "2018-09-25T07:05:47Z",
    "eventSource": "s3.amazonaws.com",
    "eventName": "PutBucketNotification",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "10.162.29.212",
    "userAgent": "signin.amazonaws.com",
    "requestParameters": {
        "notification": [
            ""
        ],
        "bucketName": "www.some-bucket.xyz",
        "NotificationConfiguration": {
            "xmlns": "http://s3.amazonaws.com/doc/2006-03-01/",
            "CloudFunctionConfiguration": {
                "CloudFunction": "arn:aws:lambda:us-east-1:61*******47:function:test-s3-change-dev-hello",
                "Event": "s3:ObjectCreated:*",
                "Id": "cbf21635-380e-4a26-92e9-de9e32f4fe7d"
            }
        }
    },
    "responseElements": null,
    "additionalEventData": {
        "vpcEndpointId": "vpce-6d72a204"
    },
    "requestID": "C5092625A5414B89",
    "eventID": "41a7b514-4d6f-41bb-974c-2864dede35eb",
    "eventType": "AwsApiCall",
    "recipientAccountId": "61*******47",
    "vpcEndpointId": "vpce-6d72a204"
}
```


A configuration change done on a Lambda Function (CloudTrail)

back to its source. This includes details on the login that initiated the 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 functionality.

CloudTrail lets you set up notifications, messages, and alerts that trigger off of configuration events in your AWS ecosystem. This means you can react to configuration errors and potential security risks as they are introduced. For example, trigger a CloudWatch Alert when there is a specific type of activity being done on an S3 bucket.

Automation is pretty critical in serverless development 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 a more traditional architecture.

How to Avoid DoS in AWS Lambda

SCALABILITY

- AWS Lambda invocation types
- Scalability considerations
- Synchronous invocations
- Asynchronous invocations
- Poll based & stream based invocations
- Poll based & not stream based invocations
- Service level mitigations
- Architectural design considerations
- Monitoring

The first thing that usually comes to mind when we think of the word “serverless” is scale. One of the biggest advantages of going serverless is that you don’t need to worry about scale or capacity planning anymore. The cloud provider does all the “heavy lifting” for you. In reality, 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 that you need to be aware of and best practices that you must follow for that to happen as planned. Otherwise, serverless applications can be vulnerable to Denial of Service attacks as any other application out there.

INVOCATION TYPES

Lambda functions can be invoked either synchronously or asynchronously. To be clear, a synchronous invocation means that the service or API that invoked the Lambda function is going to 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.

When you manually invoke a Lambda function (using either AWS CLI or AWS SDK) you can specify what invocation type you want to use,

```
--invocation-type (string)
   By  default, the Invoke API assumes RequestResponse invocation type.
   You can optionally  request  asynchronous  execution  by  specifying
   Event  as  the  InvocationType  . You can also use this parameter to
   request AWS Lambda to not execute the function but do some verifica-
   tion, such as if the caller is authorized to invoke the function and
   if the inputs are valid. You request this by  specifying  DryRun  as
   the InvocationType . This is useful in a cross-account scenario when
   you want to verify access to a function without running it.

   Possible values:

   o Event

   o RequestResponse
```

However, when you use an AWS service as a trigger, the invocation type is predetermined for each service. You have no control over the invocation type that these event sources use when they invoke your Lambda function. On the right, is a summary table, describing the different services, their invocation types and their behavior upon throttling,

Services	Event Source Type	Invocation Type	When throttled
API Gateway, Cognito, Lex, Alexa, CloudFront	Not stream based	Synchronous	**No retry built-in**.
S3, CloudWatch (Logs & Events), CloudFormation, SNS, SES, Config, CodeCommit	Not stream based	Asynchronous	**Up to 2 retries.**
DynamoDB Streams, Kinesis	Poll-based and stream based	Synchronous	Automatic retry for the failed batch of records till the data is either expired or successfully processed. The retry mechanism is **blocking**.
SQS	Poll-based and not stream based	Synchronous	Automatic retry for the failed batch of records till the Visibility Timeout period expires or the batch is processed successfully.

SCALABILITY CONSIDERATIONS

Synchronous Invocations

"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 (see note below). Each service may have its own retry policy." (AWS Documentation)

As an example, API Gateway events use synchronous invocations. An attacker that can control the amount of requests sent to API Gateway, will be able to cause throttling and as a result - Denial of Service. Applications which use synchronous invocations are easier for an attacker to target since the feedback is immediate and the attacker quickly figures out if the attack is successful or not.

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

Asynchronous Invocations

"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" (AWS Documentation).

Let's take AWS S3 as an example. An application where the user controls the frequency in which objects are uploaded to the bucket, and as a result the concurrent executions of the Lambda functions, has a potential to be throttled.

AWS states that Lambda "automatically retries the throttled event for up to six hours..." meaning that a long Denial of Service attack can eventually cause loss of data.

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

Poll Based & Stream Based Invocations

"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" (AWS Documentation)

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

Poll Based & Not Stream Based Invocations

"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." (AWS Documentation)

The messages in an SQS queue are processed, according to AWS, in the following manner - AWS Lambda automatically scales up polling activity until the number of concurrent function executions reaches 1000, the account concurrency limit, or the (optional) function concurrency limit, whichever is lower. Amazon Simple Queue Service supports an initial burst of 5 concurrent function invocations and increases concurrency by 60 concurrent invocations per minute.

Assuming we have a Lambda function that takes 5 minutes to process an event in the queue, and the account limit is 1000 concurrent executions. Since the default MessageRetentionPeriod for a message in the queue is 4 days, an attacker that attempts to cause data loss, will have to issue the following number of requests at once (assuming a DLQ is not configured):

$$\#Requests = ConcurrencyLimit \times (\frac{60}{ProcessingTime}) \times 24 \times 4$$

In our example it will be - 1,152,000. Definitely not an easy task.

MITIGATIONS AND BEST PRACTICES

Service Level Mitigations

- API Gateway provides the ability to set quota and throttling criteria.
- For relevant APIs, consider enabling API response caching, which will reduce the number of calls made to your API endpoint and also 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 as a destinations instead of a Lambda, you gain the ability to process multiple events at once (you have the ability to 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 possible inputs that might cause the function timeouts (e.g. REDOS attacks or long payloads). An attacker might be able to exploit such an application layer weakness

- PureSec Serverless Security Platform provides behavioral runtime protection against a wide range of attacks, and can reduce the risk of application layer DoS and 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 takes into account 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 won't be able to leverage them for consuming the entire account capacity.
- For Lambda functions with asynchronous event triggers (in SQS integrations - for the queue itself), set up a Dead Letter Queue. After retrying the event twice, Lambda will forward it to the DLQ destination (SQS Queue or SNS Topic) for further investigation.

Monitoring

- Monitor your Concurrent Executions account metrics. More information on how to investigate spikes in AWS Lambda concurrency can be found in this link
- Monitor your Lambda throttling metrics.
- Monitor your Lambda errors metric, specifically for timeouts
- It is highly recommend to set up monitoring & alerts on your AWS charges & billing

Using AWS Config for

COMPLIANCE

- AWS Config overview
- AWS Lambda security using config rules
- Sample config rules by PureSec

AWS CONFIG OVERVIEW

When it comes to AWS services, you can generally divide services into two classes. The operative services such as Lambda, S3, etc. and then the second class of services are helper/utility services such as CloudWatch and CloudTrail.

AWS Config belongs to the latter and is among the most useful AWS services that can dramatically improve your ability to govern your serverless applications. Governance and visibility are among the most critical component in any effective security strategy.

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

AWS Config Benefits for AWS Lambda Security:

- Continuous monitoring: monitor and record configuration or code changes of your AWS Lambda functions
- Continuous assessment: audit and assess the overall compliance of your AWS Lambda functions' 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 offers you the ability to create custom rules, associated with an AWS Lambda function. The Lambda function will contain the logic that evaluates whether your AWS resources (e.g. Lambda function) comply with the rule.

Custom rules can be invoked either in response to configuration changes or periodically (e.g. once every 24hrs), allowing them to review resources that were previously deployed to the account.

4 CONFIG RULES TO GET YOU STARTED

In the next few section, we will help you to create 4 simple Config rules, that will boost your AWS Lambda security posture. As a bonus, we already implemented the rules and uploaded them as a project to Github, including an AWS SAM application so that you will be able to deploy them quickly.

Project Github: https://github.com/puresec/lambda-config-rules

The steps for the installation are described below (and also in the Github project):

Install the dependencies:

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

Create a deployment bucket:

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

Package your application:

```
aws cloudformation package \
   --template-file template.yaml \
   --output-template-file template-packaged.yaml \
   --s3-bucket DEPLOYMENT_BUCKET_NAME
```

Deploy the Config rules:

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

Rule 1 - Detect AWS Lambda functions that are created directly through the AWS web console:

Using the AWS Lambda web console saves time when you need to experiment. We've all done that and still do it when we need to test or prototype something quickly. Having said that, that's not the proper way deploy serverless applications. Before you know it, you end up with numerous functions deployed, with no ability to restore older versions and to understand why code changed. If your organization follows CI/CD best practices, deploying new functions shouldn't be done 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 that your Lambda functions don't share the same AWS IAM execution role in order to comply with the principle of least privilege by providing each individual function the minimal amount of access required to perform its tasks. In general, this means that there should always be a 1-to-1 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 the inessential triggers, as they increase your attack surface. If your design deliberately allows a function to get invoked by multiple triggers, you should re-consider 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 standard security concept of granting the least required privileges - i.e. 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 the function will have while it executes. Under certain conditions, this permissions model, might be the thing that will save your sensitive data. With an over-permissive IAM role assigned to a function, an attacker my leverage an application layer vulnerability in your function to perform lateral movement into other resources in your AWS account.

Securing APIs With

API GATEWAY

- API types
- API keys & usage plans
- Using IAM for API access
- Amazon Cognito user pools
- Lambda custom authorizers

API GATEWAY OVERVIEW

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

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

API SECURITY

Within your system, you are likely to have APIs with different levels of access. The following section lists the different types of APIs.

Public / Unauthenticated APIs

APIs that can be consumed by any user, such as endpoints for returning a landing page which is publicly accessible and does not require users to be authenticated.

Authenticated User APIs

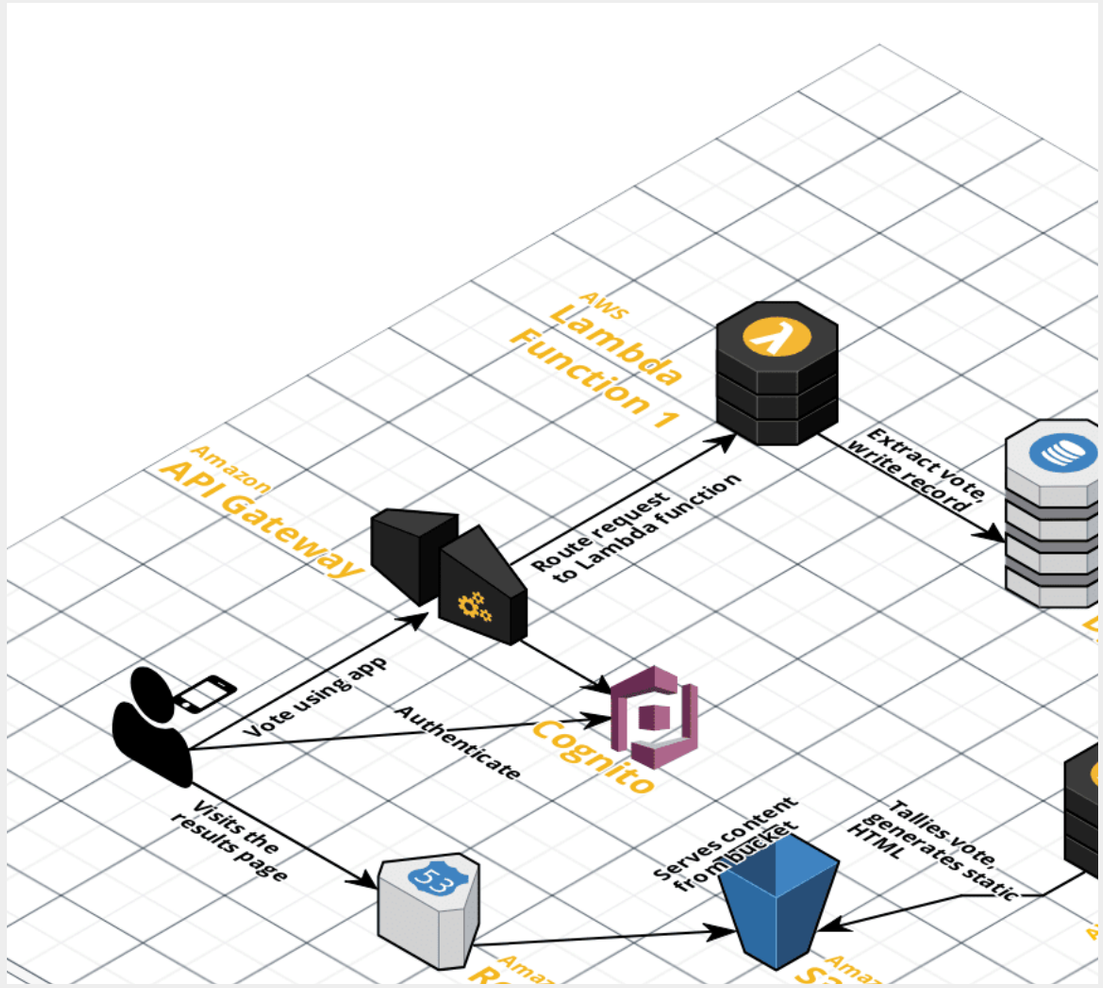
APIs that require users to be authenticated. Such APIs will typically involve updating system state on the user's behalf. For example, An API that is used to submit a vote in an online voting system.

Internal APIs

Internal APIs are meant to be used by the system itself and are not intended to be consumed by client applications directly in the micro services architecture. Such APIs are often used to encapsulate and manage a set of shared resources. For example, shared resource data that many parts of the system would need to access in a micro service 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 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



Authenticated User APIs

Gateway, you lose the ability of creating network boundaries with private VPCs. Having said that, API Gateway provides efficient access control mechanisms, which are implemented at the API gateway level. One of the ways to secure APIs with API gateway is to use API keys.

CONTROLLING ACCESS TO APIs

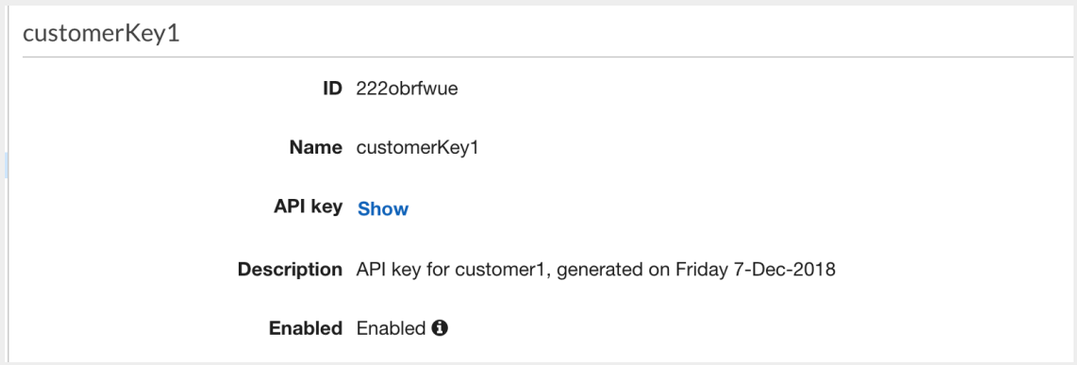
There are multiple controls in place which can be used to

control access to APIs - the most common ones are:

1. Usage plans using API keys granted to users with usage quota limiting
2. AWS IAM roles and policies
3. Amazon Cognito user pools
4. 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 that 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. API Gateway can generate API keys on your behalf, or you can import them from a CSV file.



Generating an API Key

You can find more information on how to set-up API keys for use with API Gateway in the following link

USAGE PLANS

A usage plan declares who can access one or more deployed APIs (including stages and methods), and also define the volume/frequency they can access them. The plan uses API keys to identify clients and measures access to the associated API stages for each key.

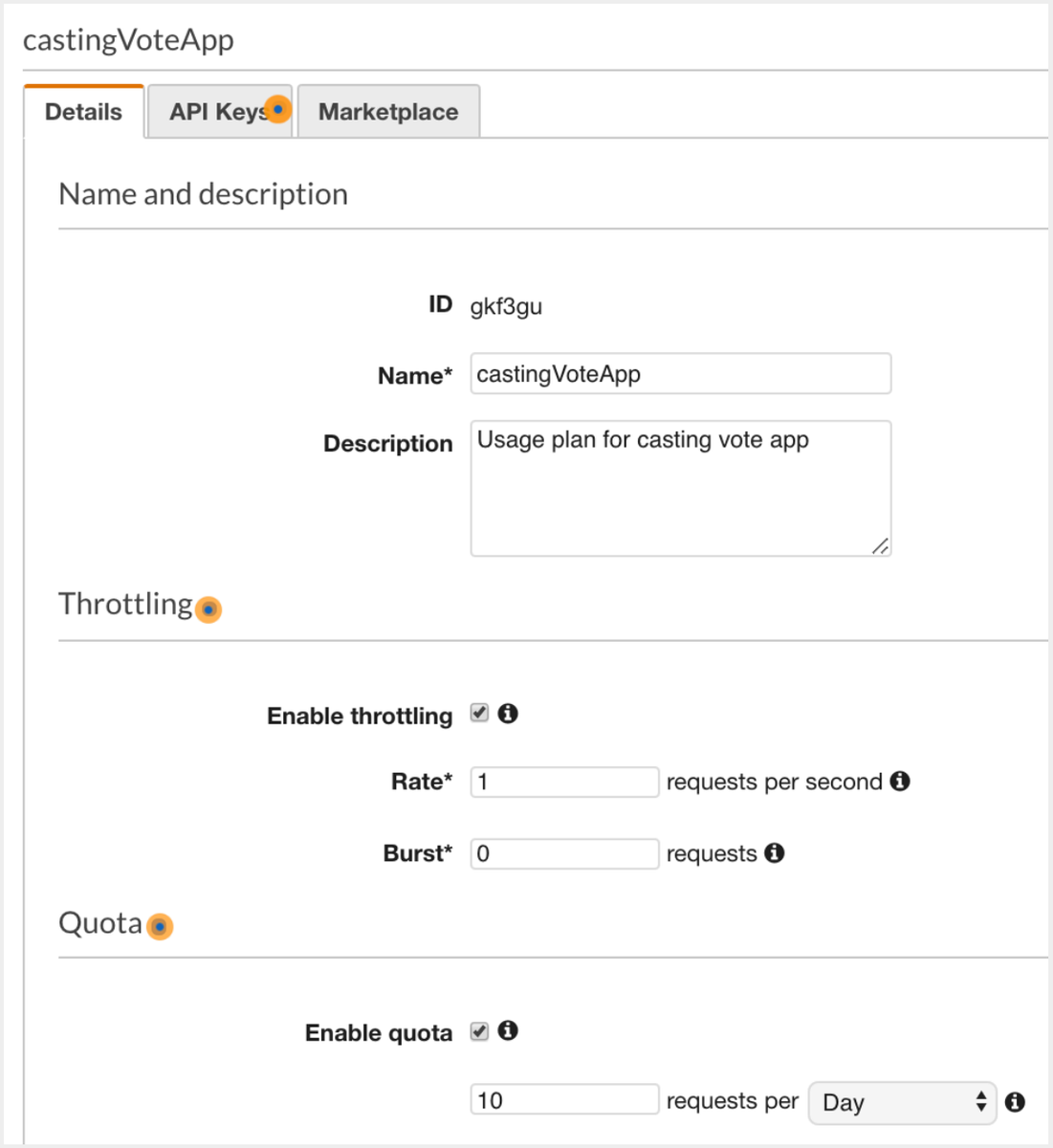
For example we might allow prospective customers to try our API during a trial period. In that case, we will limit them to a maximum of one request per second, with a request burst of up to five requests per second. At the same time, we will limit such customers to a total of no more than 10 requests per day.

More information on how to set-up usage plans can be found in the following link

CONTROLLING ACCESS TO APIs WITH AWS IAM

If you need to grant employees access to internal APIs, you would probably do so by using the AWS IAM authentication, rather than by using API keys.

To allow an API caller to invoke an API, you must first create an IAM policy that permit 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 that the caller will submit the IAM user's access keys in order to be authenticated. After setting the configuration to use the 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.



Defining an API Usage Plan with API Gateway

AMAZON COGNITO

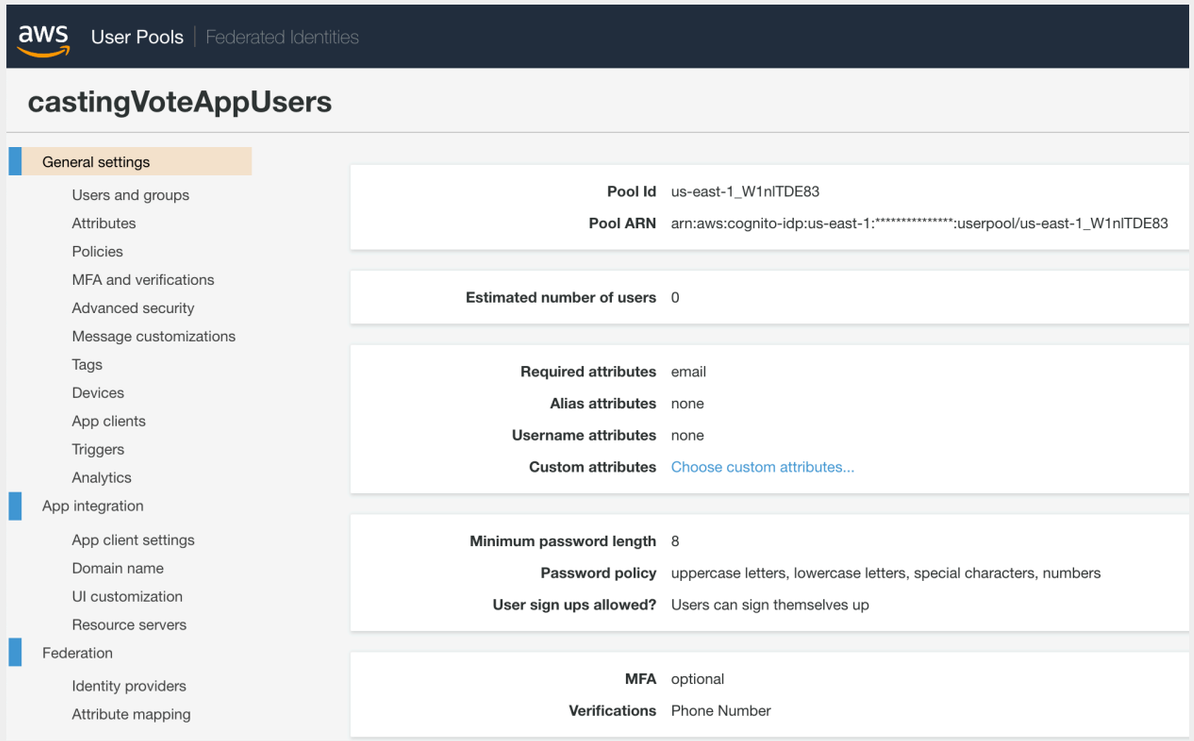
Many developers make the mistake of implementing their own authentication systems. Such systems are a good example of 'heavy lifting' work that is neither core to you business, nor something that's easy to implement correctly. In fact, many data breaches in the past were a direct result of custom built authentication systems which were poorly designed or implemented.

It is highly recommended to always prefer using an authentication service such as Amazon Cognito, Auth0 or similar. Such services remove the burden of implementing robust authentication systems, and allow 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 protocol (SRP) 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 API Gateway by authenticating them with Amazon Cognito User Pools. In such a configuration, API Gateway will validate the tokens from a successful user pool authentication, and will use them to grant your users access to resources including Lambda functions, or your APIs.



Configuring an Amazon Cognito User Pool

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 API Gateway is configured to use a Lambda authorizer, it will invoke the relevant Lambda function. During the invocation, 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

PURESEC

PureSec's Serverless Security Platform provides multi-layer threat protection for serverless applications on AWS. The platform is designed exclusively for serverless applications and defends against known & unknown attacks.

SERVERLESS APP FIREWALL & RUNTIME PROTECTION

PureSec SSP 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, PureSec's machine-learning based behavioral protection engine closely monitors function execution to detect and block unauthorized function interactions and operations, in real time.

PureSec SSP was designed from the ground up to protect serverless applications. As such, runtime protection is attached to each function, and will dynamically scale up as needed.

SERVERLESS SECURITY POSTURE

PureSec SSP seamlessly integrates into your CI/CD process. During development and build time, serverless projects are statically scanned to pinpoint risks related to over-permissive IAM roles and known vulnerable 3rd. party dependencies. With PureSec SSP integrated into your CI/CD, you are guaranteed to ship robust serverless code at all times.

SERVERLESS VISIBILITY

PureSec SSP integrates deep into your functions, providing unparalleled visibility into application layer attacks. See what your functions are doing in a way you've never seen before. For each security event, PureSec customers receive access to forensic data, allowing them to perform deep investigations into security incidents, in real time. PureSec SSP provides simple integrations with existing SIEM solutions, so your DevSecOps teams can receive event information and notifications in the tools of their choice.

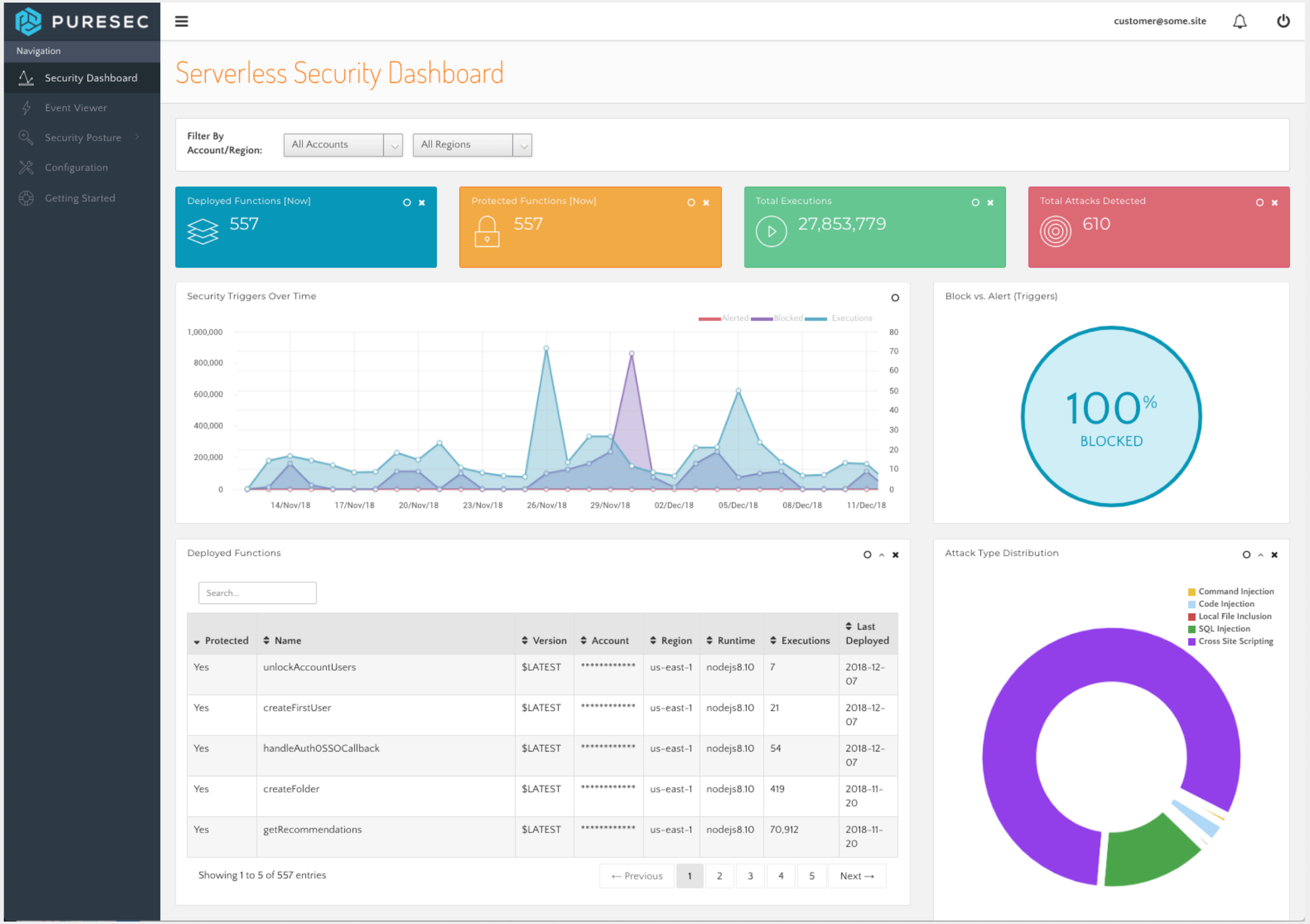
KEY PRODUCT FEATURES

Application Layer Protection:

- An event-based serverless application firewall, capable of detecting attacks such as: SQL Injection, Cross-Site Scripting, Command Injection, Runtime Code Injection, ...
- Inspection of all types of serverless event triggers
- Prevention of event input encoding attacks
- Protections against Event-Data Injection attacks

Serverless Runtime Protection:

- Malicious code behavior detection based on a pre-trained ML model.
- Hardening of the local container file system against malicious code access
- Serverless runtime environment hardening
- Outbound network access controls
- Granular function-based security policy
  Vulnerability Code Scanner:
  - Detection of over-permissive security policies
  - Detection of vulnerable 3rd party dependencies
  - Full CI/CD integration
- Dev-Ops Tools :
  - Seamless integration with cloud-native logging facilities, Splunk and other SIEMs
  - Visibility into function operations, security posture and security event information



For more information visit: https://www.puresec.io/

Links & References

APPENDIX

LINKS & REFERENCES

- Information on AWS Lambda can be found in the AWS Lambda page
- The 2018 Serverless Security Top 10 Guide
- Using IAM policies for AWS Lambda - AWS documentation page
- PureSec blog post on generating least-privileged IAM roles
- Using AWS CloudWatch with AWS Lambda documentation
- Using AWS Lambda with AWS CloudTrail documentation
- Understanding scaling behavior (AWS Lambda documentation)
- PureSec blog post on AWS Lambda DoS mitigation strategies
- PureSec blog post on using AWS Config rules for AWS Lambda
- Controlling access to an API with API Gateway (documentation)
- PureSec web site

ACKNOWLEDGEMENTS

The 'API Gateway' section of this eBook contains information contributed by Yan Cui from his online course 'Production-Ready Serverless'