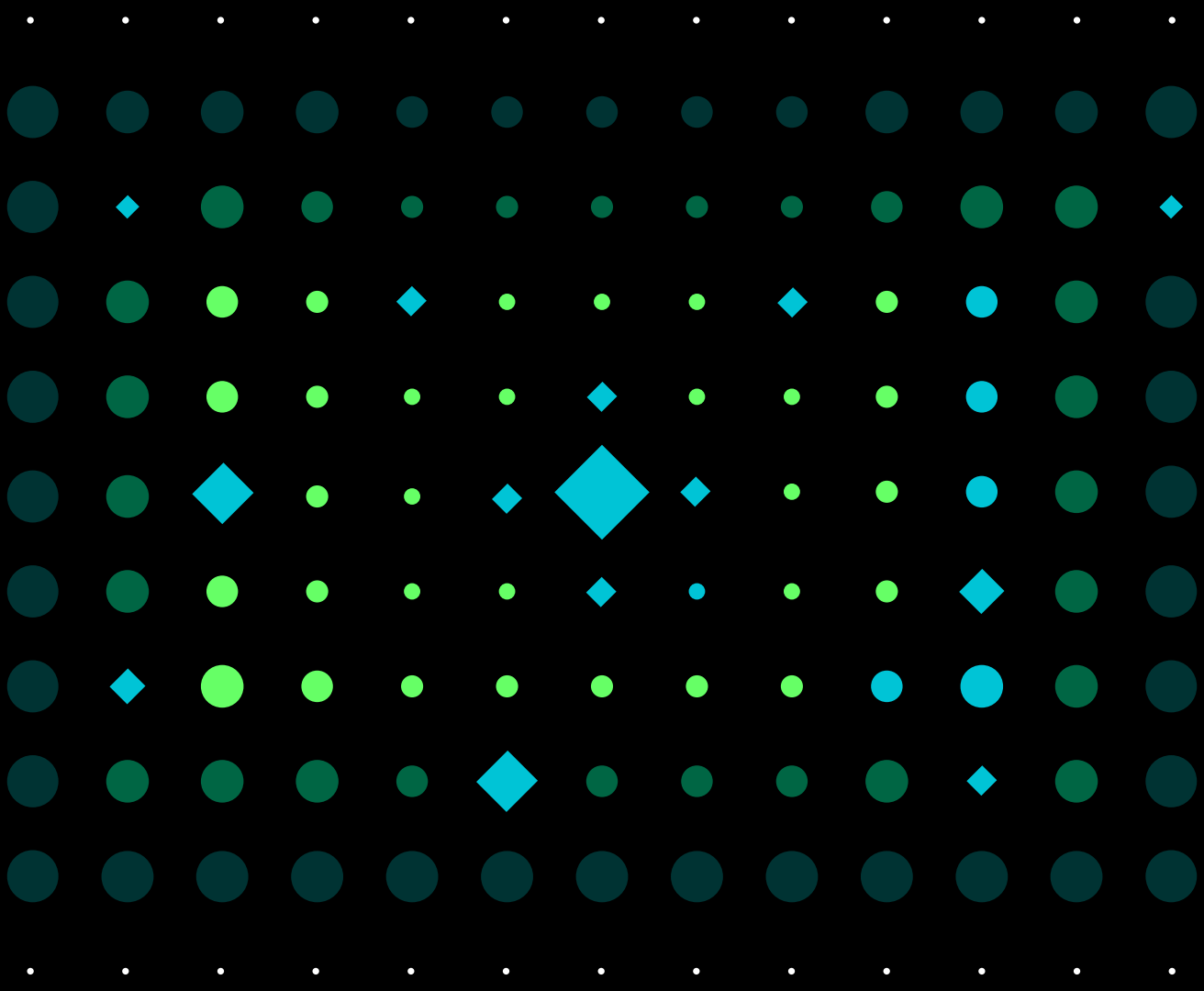
CYERA{labs}

PLATFORM WHITEPAPER

The Unexpected Threat Vector Message Queues



www.cyera.io | info@cyera.io

Table Of Contents

Summary

Cyera's security researchers continuously analyze cloud environments, looking for attack vectors that pose significant risks to data such as techniques to extract sensitive data from environments.

As part of our efforts, we discovered a popular cloud service type that poses major risks to data but is often overlooked by security teams: the message queuing service.

While there are many benefits to queues, their prevalence represents a growing data security risk. They are responsible for handling vast amounts of data, including patient, financial, and other highly sensitive data. They are unduly exposed, often housing unencrypted data or failing to log events that can alter the path of data. And the opportunities to exfiltrate sensitive queue data can come from different fronts: an insider with the right permissions or a malicious attacker with a compromised role.

Queues are fundamental to modern cloud architectures, helping applications run smoothly, achieve scalability, and improve message delivery. And their use is pervasive, ensuring efficient processing of information in the background for many of the experiences we take for granted everyday:

- **Patient care** – deliver the latest diagnosis and notes to the doctors and other healthcare providers responsible for patient recovery
- **eCommerce experience** – share website visitor interests and shopping preferences to personalize the consumer online shopping experience and increase retail sales
- **Internet of Things (IoT) functionality** – distribute device data to the appropriate databases for trend analysis or end-user devices for real-time monitoring including heart rate tracking, motion-activated camera feeds, and total miles cycled on a given day
- **Financial transactions** – route transaction information to the appropriate recipient and on time to ensure that fund transfers are successfully executed
- **Email delivery** – send emails to the recipient, timely
- **Social media** – deliver notifications to the author of a viral social posting and update the live feed

About message queues

What is AWS Simple Queue Service (SQS)

Queues are also referred to as cloud messaging services, for example: Simple Queue Service (SQS) in AWS or Azure Storage Queue (ASQ) in Azure. In this paper, we primarily use AWS examples and terminology as we examine data security risks impacting AWS SQS. However, please note that many of the scenarios and risks described below are relevant to Azure environments as well.

SQS is a fully managed message queue service offered by AWS. It enables decoupling of application components so that they can run independently and strengthen resilience against failure.

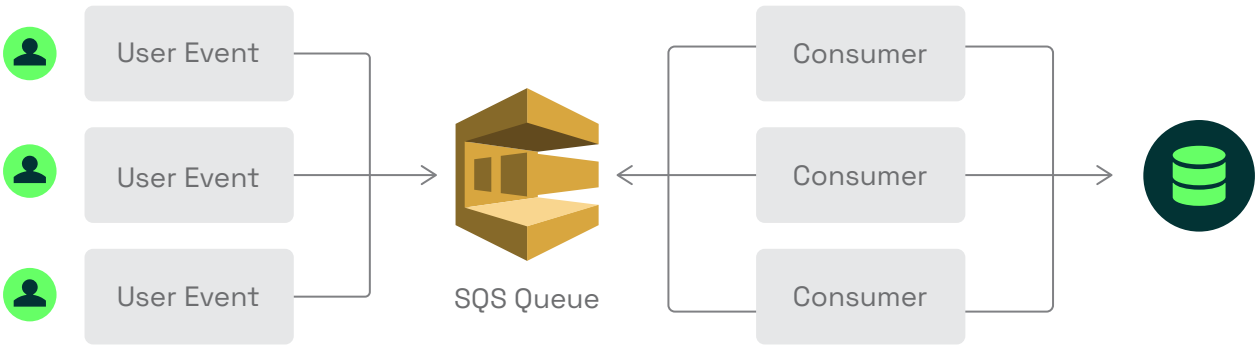
Modern applications rely on queues for intercommunication between microservices (Elastic Kubernetes Service containers, AWS Lambdas functions, etc) and to transfer data within the environment. In many cases, the data that is being transferred over queues is destined for a datastore (DynamoDB table, RDS database or S3 bucket). There are other managed messaging services in AWS, but the best known are SQS, SNS (Simple Notification Service), Kinesis, and MSK (Managed Streaming for Apache Kafka).

How AWS SQS works

The main purpose of SQS is to decouple business application logic, separate software components, and scale microservices, distributed system and serverless applications.

Let's look at a concrete use case.

A company hosts a website on AWS, generating vast amounts of traffic that is written to a database. The SQS captures this information and different consumers pull from it, polling it periodically and performing the writes to the database.



When setting up a new SQS, you may configure different options as described in the official AWS documentation.

What are the data risks of using AWS SQS?

Even though vast amounts of data are stored in S3 buckets, DynamoDB tables, Relational Database Service (RDS) databases, and data warehouses, it is vital for security teams to remember the presence of data within queues. Queues are composed of different components and layers, where data is manipulated and travels from one source to another destination. And with it, come risks of that data being mishandled or breached. Read on to understand how access to a queue's role provides unfettered access to unprotected, but highly sensitive data in transit. This includes access to toxic data combinations or opportunities to alter the recipient of data.

Exposure of sensitive data to anonymous or unauthorized entities

There is a risk of unintentionally exposing sensitive data to unauthorized users and resources. This includes when SQS are configured to allow access over the internet. When a resource-based policy enables anonymous access to data within SQS, that data is put into the path of greater risks.

IAM entities with no direct access to datastores may get access to the same underlying data via SQS. SQS often transfers data between the application's datastores or microservices in high frequency in event-driven architectures.

Data encryption through the AWS Key Management Service (KMS) services is a common method to further safeguard sensitive data in datastores. To access the data, users and resources require both permissions to the datastore and the matching KMS key. However, there is a risk that the data that was encrypted in the datastore will be accessed, unencrypted, from within the queue. Even when a resource-based policy specifies that certain sensitive data types be encrypted and access be limited to entities that have both access to that data type and the underlying KMS key, that policy can be circumvented when sensitive data is placed into the queue unencrypted, exposing it to unapproved entities.

Toxic Combinations

Toxic combinations refers to a combination of data when, put together, leads to greater risks for the organization. As a best practice, some data should be stored separately. In our research, we uncovered toxic combinations of data that, perhaps, were separated in storage but ended up in the same queue.

The following are examples of data we found intermixed into the same queues:

- Data from multiple regions
- Multi-tenant data
- Test and production data
- Employee and customer data
- Customers orders information
- Customer invoices with customer names and prices
- Billing information
- Customer names and identifiers
- API & SSH keys

Determining what types of data to combine or isolate depends on each organization's own data storage policies. For example, our research team found that one company, let's call them Company A, leveraged separate queues for each of their products. For Company B, that might not be a big deal. However for Company A, the combination of product data within a single queue presented a risk that they wanted to avoid.

Data Exfiltration

SQS is a valuable target for attackers because it stores and transports large amounts of data between distributed components. Attackers can extract sensitive data from a compromised queue. From there, they expose the queue by sending its contents directly to an external source or enabling access to it from external accounts. We will dig into this type of attack with more details below.

Queue Tampering

Attackers tamper with the queue by injecting messages and altering the messages' metadata.

The injected messages can contain content that is random or seemingly harmless. The altered metadata can make the message appear that it originated from a legitimate source.

Ultimately, the tampered messages make their way through the communication path across different resources where they could:

- disrupt business operations by triggering new, unplanned events in the services they touch
- adversely influence business decisions when they stow away into databases used as sources of truth for decision makers
- cause services to slow down or restart when invalid structure or content is introduced to the service
- open an attack path to a target system by overwhelming a critical security service, basically subjecting the queue to a DDoS attack

As you can see, the consequences impact not only data security, but also its operations and integrity.

Threat scenarios targeting default permissions

Now that we understand the risks, let's explore scenarios in which attackers can take advantage of SQS configurations to exfiltrate sensitive data. While many of the scenarios described involve an account or resource being compromised by an external attacker, the source of the compromise could come from elsewhere. In addition to external threats, malicious or negligent insiders or production error can lead to loss of sensitive data. The examples below reflect what we found in actual environments.

A wolf in sheep's clothing

Imagine you are a cloud administrator. You configure access to datastores and limit which service accounts have access to sensitive datastores. Should attackers compromise a service with access to sensitive datastores, we all know that would be bad. What if the attackers compromises a service WITHOUT access to sensitive datastores. Then the risk is lessened..., right?

As it turns out, the service WITHOUT access to sensitive datastores, though seemingly inconsequential on the surface, has enabled attackers to access sensitive data via the queue.

In one AWS environment we examined, we found that SQS had stored messages containing customer order history that include customer identifiers, merchant names, prices, and more, all considered sensitive data. We identified two Lambda functions that did not have access to the S3 buckets, DynamoDB tables, or other datastores containing sensitive data. Yet, those Lambda functions had access to receive all SQS messages and consequently the sensitive data contained in those messages. We recognized that the same sensitive data flows through the queues and is stored in the restricted DynamoDB tables.

The Lambdas had a role named "APIGatewayLambdaExecRole", governed by the IAM policy "APIGatewayLambdaExecPolicy."

```
"Arn:"arn:aws: iam:: <REDACTED_ACCOUNT_NUMBER>: policy/APIGatewayLambdaExecPolicy"
      "AttachmentCount": 1,
      “CreateDate”: “2016-02-04 15:17:23+00:00”,
      "DefaultVersionId": "v2",
      "IsAttachable": true,
      "Path":"/",

      "Permissions BoundaryUsageCount": 0,
      “PolicyId”: <REDACTED>
      "PolicyName": "APIGatewayLambdaExecPolicy",
      "PolicyVersionList":[
          {
              "CreateDate": "2016-02-04 15:26:05+00:00",
              "Document":  {
                      "Statement": [
                        {
                          "Action": [
                              "logs:*"
                          [  ,
                          "Effect": "Allow",
                          “Resource”: “arn: aws: logs:*:*:*”
                        },
                        {
                          "Action": [
                              "sqs:*"
                          ],
                          "Effect": "Allow",
                          "Resource":"arn:aws:sqs:*:*:*"
                        }
                      ],
                      "Version": "2012-10-17"
              },
              "IsDefaultVersion": true,
              "VersionId": "v2"
          }
```

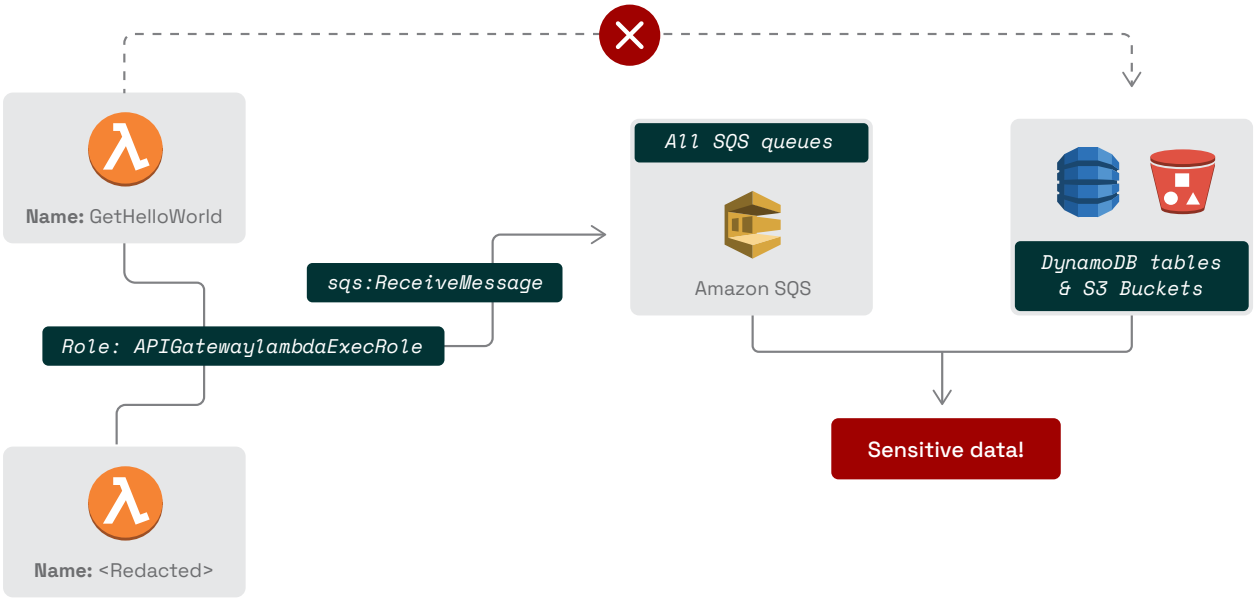
Note this role provided these Lambda functions:

sqs:* permissions, on resource - “arn:aws:sqs:*:*:*"

This means the Lambda functions have `ReceiveMessage` permissions for all SQS in the account, even though they have no access to any other data-related resource! As the name of the permissions suggests, `ReceiveMessage` permissions provide the functions with the ability to pull messages from the queue.

Not only were these Lambda functions assigned overly permissive access, their level of access was assigned by default in AWS. We therefore expect that organizations may be unaware of both the prevalence and risks of such configurations in their own AWS environments.

In the aforementioned account, this is how the Lambda function, “GetHelloWorld” accesses sensitive data in the SQS.



In examining actual environments, we noted that SNS configurations can be similarly targeted. As mentioned before, SNS and SQS are commonly used together in distributed application architectures. Often, SQS are subscribed to SNS topics to receive messages. New subscribers can be added to SNS topics, using the SNS:Subscribe IAM permission, and receive messages that are subsequently published to the topic.

If an account with the SNS:Subscribe permission is compromised, attackers can add subscribers to the SNS topic. These subscribers may be an external email address, HTTP endpoint, Lambda function, SQS, and more. Similar to the scenario above, attackers gain access to sensitive data despite not having compromised an account with access to sensitive datastores.

The next question that comes to mind is how attackers can compromise the Lambda function that contains permissions to receive messages from all queues.

Let's take a look at the attack flow:

1. Attackers compromise an account with the following permissions on one of the Lambda functions: UpdateFunctionCode and InvokeFunction. As a side note, we discovered a number of Lambda function roles that had this permission combination.
2. Attackers use the combination of these two permissions to initiate an attack.
3. Attackers update the Lambda function code to enumerate all queues.
4. He sends the content of the SQS messages with the sensitive data to an external, controlled machine or server.

While we provide an example of an external attacker, an insider such as an employee could follow a similar attack flow.

Note that even though our analysis focused on Lambda functions with access to SQS, we recognized that other AWS services can similarly be compromised: an AWS service that does not have access to any data-related resource could, at the same time, receive messages with sensitive data from queues.

Exposure through a third-party

In another notable example, we identified the well-known third-party Kubernetes Autoscaler service with `ReceiveMessage` permissions to all queues in an account. This was the default configuration as part of the setup and integration of this service in the account.

The Kubernetes Autoscaler requires the `ReceiveMessage` permission to determine the performance and latency of resources for autoscaling decisions. The service measures the time it takes to receive messages from the queue, allowing it to process sensitive data. Should attackers compromise the service, they will be able to read the sensitive data.

To mitigate this issue, you should either restrict the service's access to specific queues that are relevant for resource scaling or use client-side encryption when storing messages in queues. This allows the service to make autoscaling decisions while limiting its access to sensitive data or preventing it from exposing the underlying content as plaintext.

Services with unlimited resource access to SQS

As part of our research, we were intrigued by other AWS services with unlimited resource access.

One interesting AWS service that fit this profile was the Elastic Beanstalk service:

```
aws-elasticbeanstalk-ec2-role
Permissions:
[“sqs:ChangeMessageVisibility”,“sqs:DeleteMessage”,“sqs:ReceiveMessage”,“sqs:
SendMessage”]

Queues: ALL
```

Whenever Elastic Beanstalk is created in an AWS account, it creates a default instance profile and assigns managed policies with default permissions to it.

The default permissions include SQS `SendMessage` and SQS `ReceiveMessage` to all queues in the account as part of the AWSElasticBeanstalkWorkerTier managed policy. The SQS statements provide unlimited queues resource access, while other statements that provide access to datastores are limited only to specific resources, such as buckets with names that contain `elasticbeanstalk` or DynamoDB tables with names that contain `stack-AWSEBWorkerCronLeaderRegistry`.

And there are other AWS services with default permissions that grant unlimited SQS access:

- AmazonEC2RoleforDataPipelineRole
- EMR_EC2_DefaultRole

In addition to the two AWS examples, we encountered third-party services that integrated with AWS and required the excessive `ReceiveMessage` permission to all queues in the account.

Should attackers compromise just one of these services, they would gain access to the sensitive data from all queues, thus multiplying the exposure of sensitive data exfiltration risks. It is therefore strongly recommended to review the default permissions that are associated with new resources across not only AWS, but your other cloud providers and verify that the permissions for accessing sensitive data is both consistent and well understood across resources.

Default policies:

- AWSLambdaSQSQueueExecutionRole

```
{
      “Version”: “2012-10-17”,
      “Statement”: [
              {
              “Effect”: “Allow”,
              “Action”: [
                          “sqs:ReceiveMessage”,
                          “sqs:DeleteMessage”,
                          “sqs:GetQueueAttributes”,
                          “logs:CreateLogGroup”,
                          “Logs:CreateLogStream”,
                          “Logs:PutLogEvents”
              ]
              “Resource”: “*”
            }
      ]
}
```

Inconsistent isolation

One common architectural approach is to isolate environment data types – staging and production – by placing them into different accounts. It is considered a best practice, allowing the roles within the account to compliantly access the data they are authorized to and reducing opportunities for the roles to access dataa outside of what is authorized.

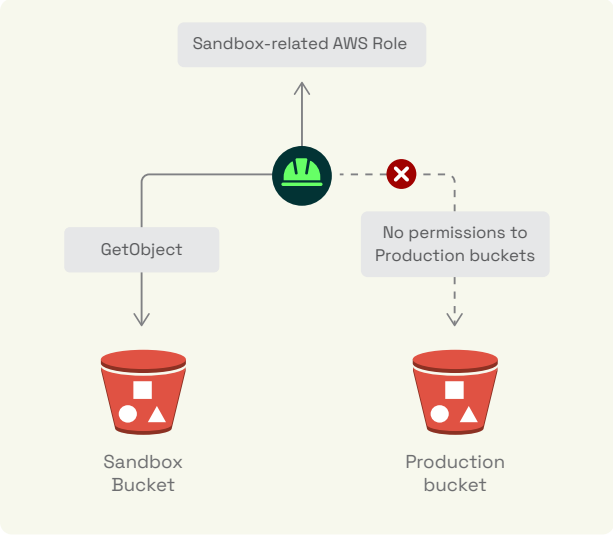
However, some admins enforce isolation by using a single AWS account to store both production and staging data. Then, they put each environment data type into its own datastores. For example, production data goes into a production-dedicated bucket and staging data goes into a staging-dedicated bucket.

Another isolation approach is to group workloads, roles, and datastores together by environment data type. This is best illustrated by the following table:

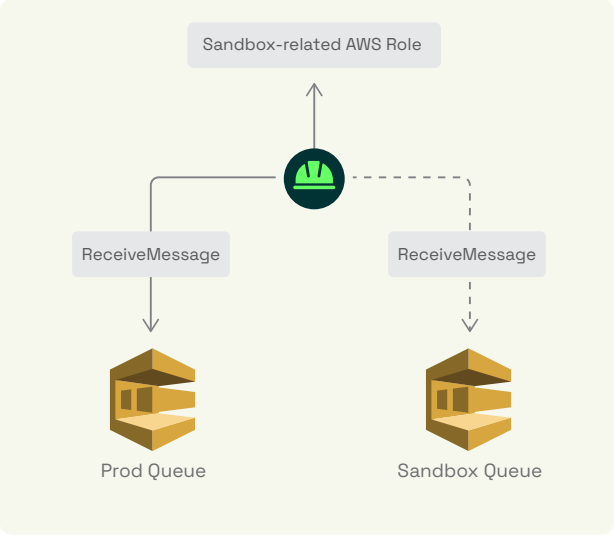
Staging data	Production data
Workload A	Workload B
Role A	Role B
Datastore A	Datastore B

As you can see, admins dedicated different workloads, roles, and datastores for different environment data types. And the workloads, roles, and datastores do not have access to any other environment data types.

In one environment that we examined, we noticed that the admin enforced isolation at a lower management tier but that the approach was applied inconsistently. The admin dedicated one set of workloads, roles, and datastores for sandbox data and another set for production data. However, a role, which was supposed to have access only to sandbox data, had access to both production and sandbox queues.



A role with access permissions to the sandbox bucket but not the production bucket



A role with access permissions to both production and sandbox queues

Under the radar data exfiltration

From the attackers' perspective, the goal is to steal large amounts of data without getting caught. It certainly helps when their actions are not logged or, even when logged, go unnoticed.

SQS `ReceiveMessage` events are not logged via the standard logging service, CloudTrail, since the service does not log dataplane events. While some services, such as S3, allows you to log dataplane events such as **S3:GetObject**, SQS does not support this.

Due to the lack of logging, it will be difficult to detect an attack that targets SQS. When detected, it will be difficult to trace the root cause of the attack.

However, via CloudWatch metrics, you can look for suspicious events such as a sudden spike in `NumberOfMessagesReceived`, which shows the number of messages received over time from specific queues.

Other permissions can be targeted by attackers to gain control of the queue. These permissions can be combined with the `ReceiveMessage` permissions to aid in data exfiltration efforts:

- `SetQueueAttributes` - enables users (or the attacker) to update the queue's permissions. SQS supports resource-based policies, which can make the queue publicly accessible or accessible to specific, external entities. Making publicly accessible a resource such as S3 is generally blocked by the Block Public Access feature, which is commonly applied to resources across AWS. This feature does not exist for SQS.
- `AddPermission` - enables users (or the attacker) to update the queue's policies and enables sharing of access to the queue with specific entities.
- `TagQueue` - enables users (or the attacker) to add tags to the queue. With tag-based access policies, users are authorized to view data with certain tags only. By changing or adding new tags, attackers decide who can access the data. Should attackers compromise a user with access rights to certain tags, then attackers are provided a means to gain access to the data.

Let's say attackers compromise a role with full permissions to SQS, such as one of the roles mentioned above. Attackers can covertly compromise this role to exfiltrate data via the following steps:

1. Update one of the queue's policy to enable external access or internet access via the `SetQueueAttributes` or `AddPermission` permission or create a new queue
2. Send the sensitive data to the queue (`SendMessage` logs are not being monitored via CloudTrail)
3. Use the queue's name (Amazon Resource Name) to send messages to an external account

Note that other messaging services such as SNS may also be targeted in a similar manner. Attackers can publicly expose the SNS topic and subscribe an external source to receive its messages.

Missing encryption and unintended access

In some environments, we found that SQS was encrypted via KMS keys, using either AWS or customer-managed keys. KMS was used to further restrict access to queues based on the data in the queues and apply multiple access boundaries on the sensitive data. To access KMS-encrypted SQS, the role must have both the IAM permissions to the SQS resource and KMS permissions to the KMS key.

Here are some examples of how SQS and KMS were utilized for different types of data and services:

- **Production data** – SQS encrypted with KMS
- **Non-production data** – SQS are not encrypted
- **Application data** – dedicated SQS and KMS key for each type of application data
- **Microservice** – dedicated SQS and KMS key for each microservice

We identified inconsistencies in how KMS encryption was applied with sensitive data:

- **Sensitive data** – SQS encrypted with KMS
- **The same type of sensitive data** – SQS not encrypted

This may lead to unauthorized and unintended access to sensitive data by entities that do not have access to the encryption key, but have access via the IAM permissions.

Triggering data deletion by parameters injection

Not only are queues used to transfer data between services, they are used to trigger events in other services such as to create, delete, or duplicate data to other destinations.

In some environments, we found that parameters inside SQS messages triggered data deletion as part of the application flow. We encountered a Lambda function that accepted new client information and sent messages to the SQS to trigger deletion for specific customer forms. Those forms contained the fields “customer_id” and “form_id”.

Let’s look at the code snippet from the Lambda function (modified and shortened for readability):

```
def delete_customer_form(_: message: SQSMessage, sqs_client: SQSClient) -> None:
    form_id = message.body.get(“form_id”)
              form_parameters_validation(form_id)
    logger.info(f”Sending message to delete form SQS with form {form_id}.”)
    sqs_client.send_message(
        f”Delete form {form_id}.”,
        DELETE_CUSTOMER_DATA_QUEUE_URL,
        customer_id=message.customer_id,
        form_id=form_id,
)
```

The messages from this deletion queue were received by another Lambda function that performed the deletion of all the customer form-related data from backend datastores in S3 buckets and DynamoDB tables.

Despite there being a parameter validation in place on the send side, the Lambda function on the receiving end did not have the same validation in place. It consumes the message from the deletion queue, extracts the "customer_id" and "form_id" parameter from each message, and performs the matching lookups in S3 bucket files and databases.

```
def delete_customer_form(event: Dict[str, Any], context: Any) -> None:
    messages = event.get("Records", [])

    for message in messages:
        form_id = get_form_attribute(message, "form_id")
        customer_id = get_customer_attribute(message, "customer_id")
        logger.info(
            f"Starting to delete form for customer_id=`{customer_id}` "
            f"and form_id=`{form_id}`"
        )
  delete_form(customer_id,form_id)
```

Attackers with permission to send messages to the queue could enumerate the "form_id" and "customer_id" fields and send messages to the queue. This would cause the Lambda function to delete customer data from the environment's datastores. Notice that many of the attack scenarios described in this paper result in data exfiltration; however, this scenario results in deletion of data. The attacker may have malicious intent to harm the target victim immediately, without necessarily stealing the data for future sale or leverage.

Crawling through queue names

In one of the environments we analyzed, we found that admins used dedicated queues for each tenant. This method provides the highest level of isolation and security, though with higher costs and operational complexity. The admins assigned a name to each queue, utilizing a naming convention that included the names of their customers. This resulted in a predictable naming pattern across distinct queues.

One of the APIs used in SQS is the `get-queue-attributes` API, which receives a queue-url parameter and returns details about the queue, such as creation date, message retention period, and approximate number of messages in the queue.

We noticed that the API returns different responses based on whether the queried queue URL existed or not, even when it did not have permissions to access the queue.

Querying existing queue which we don't have access to:

```
aws sqs get-queue-attributes --queue-url "https://queue.amazonaws.com/<ACCOUNT_I
D>/<RESTRICTED_QUEUE_NAME>
```

returns:

```
An error occurred (AccessDenied) when calling the GetQueueAttributes operation:
Access to the resource https://queue.amazonaws.com/ is denied.
```

While querying non-existent queues returns:

```
An error occurred (AWS.SimpleQueueService.NonExistentQueue) when calling the Get
QueueAttributes operation: The specified queue does not exist or you do not have
access to it.
```

When the SQS naming convention is discovered, an attacker may crawl through queue names, testing out which ones work to reveal the identities of the target company's customers. Any authenticated AWS account can deploy this method to discover non-public information, especially if the account is controlled by a former employee with internal knowledge of how the queues were named.

We need to use a profile from the same region as the queue's:

```
aws sqs get-queue-attributes --queue-url "https://us-west-2.queue.amazonaws.com/
<ACCOUNT_NUMBER>/<RESTRICTED_QUEUE>
```

If the queue has a us-west-2 address, then we need to set the profile to `-region us-west-2` so it would generate different responses between existing and non-existing queues as demonstrated above.

Conclusion

Sensitive data moves across the cloud environment, depending on queues with different risk exposure levels to store and transmit data to other resources. With such widespread reliance on queues, it is crucial to have visibility into the types of sensitive data that is pushed through the queues. It is equally important to ensure that consistent controls are in place for all the resources and services that handle sensitive data.

We found that many organizations were unaware of the risks of leaving default roles in place. Simply going with default roles from cloud service providers is not an adequate data security strategy. Hardening security posture in cloud environments requires understanding what those default configurations are and how they impact both functionality or security.

We recommend that you evaluate the permissions assigned to queues, checking whether default permissions expose sensitive data. When possible, we advise that you follow the principle of least privilege when configuring access permissions. If your organization has an encryption policy for storing certain types of sensitive data, then a next step is to evaluate if those policies follow the sensitive data as it moves downstream. When data moves from source to destination and all the microservices in between, it is critical to ensure that your policies are applied consistently.

About Cyera

Cyera is the data security company that gives businesses deep context on their data, applying correct, continuous controls to assure cyber-resilience and compliance. Cyera takes a data-centric approach to security across your data landscape, empowering security teams to know where their data is, what exposes it to risk, and take immediate action to remediate exposures. Backed by leading investors including Sequoia, Accel, and Cyberstarts, Cyera is redefining the way companies do cloud data security. To learn more, visit www.cyera.io.

Trusted by:

Cboe LifeLabs MERCURY FINANCIAL UTA ACV AUCTIONS