AWS Compute Blog

Implementing AWS Well-Architected best practices for Amazon SQS – Part 2

by Pascal Vogel | on 27 JUN 2023 | in Amazon Simple Queue Service (SQS), AWS Well-Architected Framework, Serverless | Permalink | Share

This blog is written by Chetan Makvana, Senior Solutions Architect and Hardik Vasa, Senior Solutions Architect.

This is the second part of a three-part blog post series that demonstrates implementing best practices for Amazon Simple Queue Service (Amazon SQS) using the AWS Well-Architected Framework.

This blog post covers best practices using the Security Pillar and Reliability Pillar of the AWS Well-Architected Framework. The inventory management example introduced in part 1 of the series will continue to serve as an example.

See also the other two parts of the series:

- Implementing AWS Well-Architected Best Practices for Amazon SQS – Part 1: Operational Excellence
- Implementing AWS Well-Architected Best Practices for Amazon SQS – Part 3: Performance Efficiency, Cost Optimization, and Sustainability

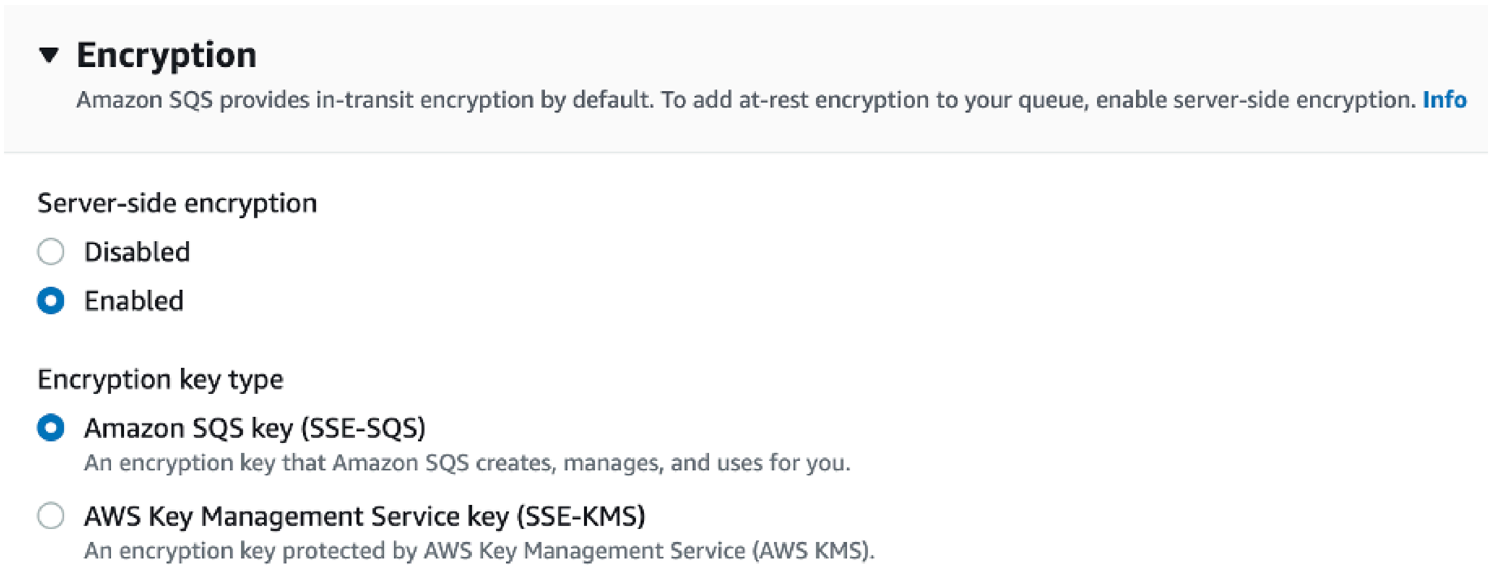
Security Pillar

The Security Pillar includes the ability to protect data, systems, and assets and to take advantage of cloud technologies to improve your security. This pillar recommends putting in place practices that influence security. Using these best practices, you can protect data while in-transit (as it travels to and from SQS) and at rest (while stored on disk in SQS), or control who can do what with SQS.

Best practice: Configure server-side encryption

If your application has a compliance requirement such as HIPAA, GDPR, or PCI-DSS mandating encryption at rest, if you are looking to improve data security to protect against unauthorized access, or if you are just looking for simplified key management for the messages sent to the SQS queue, you can leverage Server-Side Encryption (SSE) to protect the privacy and integrity of your data stored on SQS.

SQS and AWS Key Management Service (KMS) offer two options for configuring server-side encryption. SQS-managed encryptions keys (SSE-SQS) provide automatic encryption of messages stored in SQS queues using AWS-managed keys. This feature is enabled by default when you create a queue. If you choose to use your own AWS KMS keys to encrypt and decrypt messages stored in SQS, you can use the SSE-KMS feature.


▼ **Encryption**
Amazon SQS provides in-transit encryption by default. To add at-rest encryption to your queue, enable server-side encryption. Info

Server-side encryption
- ○ Disabled
- ● Enabled

Encryption key type
- ● Amazon SQS key (SSE-SQS)
  An encryption key that Amazon SQS creates, manages, and uses for you.
- ○ AWS Key Management Service key (SSE-KMS)
  An encryption key protected by AWS Key Management Service (AWS KMS).

SSE-KMS provides greater control and flexibility over encryption keys, while SSE-SQS simplifies the process by managing the encryption keys for you. Both options help you protect sensitive data and comply with regulatory requirements by encrypting data at rest in SQS queues. Note that SSE-SQS only encrypts the message body and not the message attributes.

In the inventory management example introduced in part 1, an AWS Lambda function responsible for CSV processing sends incoming messages to an SQS queue when an inventory updates file is dropped into the Amazon Simple Storage Service (Amazon S3) bucket. SQS encrypts these messages in the queue using SQS-SSE. When a backend processing Lambda polls messages from the queue, the encrypted message is decrypted, and the function inserts inventory updates into Amazon DynamoDB.

The AWS Could Development Kit (AWS CDK) code sets SSE-SQS as the default encryption key type. However, the following AWS CDK code shows how to encrypt the queue with SSE-KMS.

Python
```python
# Create the SQS queue with DLQ setting
queue = sqs.Queue(
    self,
    "InventoryUpdatesQueue",
    visibility_timeout=Duration.seconds(300),
    encryption=sqs.QueueEncryption.KMS_MANAGED,
)
```

Best practice: Implement least-privilege access using access policy

For securing your resources in AWS, implementing least-privilege access is critical. This means granting users and services the minimum level of access required to perform their tasks. Least-privilege access provides better security, allows you to meet your compliance requirements, and offers accountability via a clear audit trail of who accessed what resources and when.

By implementing least-privilege access using access policies, you can help reduce the risk of security breaches and ensure that your resources are only accessed by authorized users and services. AWS Identity and Access Management (IAM) policies apply to users, groups, and roles, while resource-based policies apply to AWS resources such as SQS queues. To implement least-privilege access, it's essential to start by defining what actions are required for each user or service to perform their tasks.

In the inventory management example, the CSV processing Lambda function doesn't perform any other task beyond parsing the inventory updates file and sending the inventory records to the SQS queue for further processing. To ensure that the function has the permissions to send messages to the SQS queue, grant the SQS queue access to the IAM role that the Lambda function assumes. By granting the SQS queue access to the Lambda function's IAM role, you establish a secure and controlled communication channel. The Lambda function can only interact with the SQS queue and doesn't have unnecessary access or permissions that might compromise the system's security.

Python
```python
# Create pre-processing Lambda function
csv_processing_to_sqs_function = _lambda.Function(
    self,
    "CSVProcessingToSQSFunction",
    runtime=_lambda.Runtime.PYTHON_3_8,
    code=_lambda.Code.from_asset("sqs_blog/lambda"),
    handler="CSVProcessingToSQSFunction.lambda_handler",
    role=role,
    tracing=Tracing.ACTIVE,
)

# Define the queue policy to allow messages from the Lambda function's role only
policy = iam.PolicyStatement(
    actions=["sqs:SendMessage"],
    effect=iam.Effect.ALLOW,
    principals=[iam.ArnPrincipal(role.role_arn)],
    resources=[queue.queue_arn],
)

queue.add_to_resource_policy(policy)
```

Best practice: Allow only encrypted connections over HTTPS using aws:SecureTransport

It is essential to have a secure and reliable method for transferring data between AWS services and on-premises environments or other external systems. With HTTPS, a network-based attacker cannot eavesdrop on network traffic or manipulate it, using an attack such as man-in-the-middle.

With SQS, you can choose to allow only encrypted connections over HTTPS using the aws:SecureTransport condition key in the queue policy. With this condition in place, any requests made over non-secure HTTP receive a 400 InvalidSecurity error from SQS.

In the inventory management example, the CSV processing Lambda function sends inventory updates to the SQS queue. To ensure secure data transfer, the Lambda function uses the HTTPS endpoint provided by SQS. This guarantees that the communication between the Lambda function and the SQS queue remains encrypted and resistant to potential security threats.

Python
```python
# Create an IAM policy statement allowing only HTTPS access to the queue
secure_transport_policy = iam.PolicyStatement(
    effect=iam.Effect.DENY,
    actions=["sqs:*"],
    resources=[queue.queue_arn],
    conditions={
        "Bool": {
            "aws:SecureTransport": "false",
        },
    },
)
```

Best practice: Use attribute-based access controls (ABAC)

Some use-cases require granular access control. For example, authorizing a user based on user roles, environment, department, or location. Additionally, dynamic authorization is required based on changing user attributes. In this case, you need an access control mechanism based on user attributes.

Attribute-based access controls (ABAC) is an authorization strategy that defines permissions based on tags attached to users and AWS resources. With ABAC, you can use tags to configure IAM access permissions and policies for your queues. ABAC hence enables you to scale your permission management easily. You can author a single permission policy in IAM using tags created for each business role, and no longer need to update the policy when adding new resources.

ABAC for SQS queues enables two key use cases:

- Tag-based access control: use tags to control access to your SQS queues, including control plane and data plane API calls.
- Tag-on-create: enforce tags at the time of creation of an SQS queues and deny the creation of SQS resources without tags.

Reliability Pillar

The Reliability Pillar encompasses the ability of a workload to perform its intended function correctly and consistently when it's expected to. By leveraging the best practices outlined in this pillar, you can enhance the way you manage messages in SQS.

Best practice: Configure dead-letter queues

In a distributed system, when messages flow between sub-systems, there is a possibility that some messages may not be processed right away. This could be because of the message being corrupted or downstream processing being temporarily unavailable. In such situations, it is not ideal for the bad message to block other messages in the queue.

Dead Letter Queues (DLQs) in SQS can improve the reliability of your application by providing an additional layer of fault tolerance, simplifying debugging, providing a retry mechanism, and separating problematic messages from the main queue. By incorporating DLQs into your application architecture, you can build a more robust and reliable system that can handle errors and maintain high levels of performance and availability.

In the inventory management example, a DLQ plays a vital role in adding message resiliency and preventing situations where a single bad message blocks the processing of other messages. If the backend Lambda function fails after multiple attempts, the inventory update message is redirected to the DLQ. By inspecting these unconsumed messages, you can troubleshoot and redrive them to the primary queue or to custom destination using the DLQ redrive feature. You can also automate redrive by using a set of APIs programmatically. This ensures accurate inventory updates and prevents data loss.

The following AWS CDK code snippet shows how to create a DLQ for the source queue and sets up a DLQ policy to only allow messages from the source SQS queue. It is recommended not to set the `max_receive_count` value to 1, especially when using a Lambda function as the consumer, to avoid accumulating many messages in the DLQ.

Python
```python
# Create the Dead Letter Queue (DLQ)
dlq = sqs.Queue(self, "InventoryUpdatesDlq", visibility_timeout=Duration.seconds(300))

# Create the SQS queue with DLQ setting
queue = sqs.Queue(
    self,
    "InventoryUpdatesQueue",
    visibility_timeout=Duration.seconds(300),
    dead_letter_queue=sqs.DeadLetterQueue(
        max_receive_count=3,  # Number of retries before sending the message to the DL
        queue=dlq,
    ),
)
# Create an SQS queue policy to allow source queue to send messages to the DLQ
policy = iam.PolicyStatement(
    effect=iam.Effect.ALLOW,
    actions=["sqs:SendMessage"],
    resources=[dlq.queue_arn],
    conditions={"ArnEquals": {"aws:SourceArn": queue.queue_arn}},
)
queue.queue_policy = iam.PolicyDocument(statements=[policy])
```

Best practice: Process messages in a timely manner by configuring the right visibility timeout

Setting the appropriate visibility timeout is crucial for efficient message processing in SQS. The visibility timeout is the period during which SQS prevents other consumers from receiving and processing a message after it has been polled from the queue.

To determine the ideal visibility timeout for your application, consider your specific use case. If your application typically processes messages within a few seconds, set the visibility timeout to a few minutes. This ensures that multiple consumers don't process the message simultaneously. If your application requires more time to process messages, consider breaking them down into smaller units or batching them to improve performance.

If a message fails to process and is returned to the queue, it will not be available for processing again until the visibility timeout period has elapsed. Increasing the visibility timeout will increase the overall latency of your application. Therefore, it's important to balance the tradeoff between reducing the likelihood of message duplication and maintaining a responsive application.

In the inventory management example, setting the right visibility timeout helps the application fail fast and improve the message processing times. Since the Lambda function typically processes messages within milliseconds, a visibility timeout of 30 seconds is set in the following AWS CDK code snippet.

Python
```python
queue = sqs.Queue(
    self,
    " InventoryUpdatesQueue",
    visibility_timeout=Duration.seconds(30),
)
```

It is recommended to keep the SQS queue visibility timeout to at least six times the Lambda function timeout, plus the value of `MaximumBatchingWindowInSeconds`. This allows Lambda function to retry the messages if the invocation fails.

Conclusion

This blog post explores best practices for SQS using the Security Pillar and Reliability Pillar of the AWS Well-Architected Framework. We discuss various best practices and considerations to ensure the security of SQS. By following these best practices, you can create a robust and secure messaging system using SQS. We also highlight fault tolerance and processing a message in a timely manner as important aspects of building reliable applications using SQS.

The next part of this blog post series focuses on the Performance Efficiency Pillar, Cost Optimization Pillar, and Sustainability Pillar of the AWS Well-Architected Framework and explore best practices for SQS.

For more serverless learning resources, visit Serverless Land.

TAGS: contributed, serverless