User Guide

AWS Secrets Manager

AWS Secrets Manager: User Guide

Table of Contents

What is AWS Secrets Manager?

AWS Secrets Manager helps you manage, retrieve, and rotate database credentials, application credentials, OAuth tokens, API keys, and other secrets throughout their lifecycles. Many AWS services store and use secrets in Secrets Manager.

Secrets Manager helps you improve your security posture, because you no longer need hard-coded credentials in application source code. Storing the credentials in Secrets Manager helps avoid possible compromise by anyone who can inspect your application or the components. You replace hard-coded credentials with a runtime call to the Secrets Manager service to retrieve credentials dynamically when you need them.

With Secrets Manager, you can configure an automatic rotation schedule for your secrets. This enables you to replace long-term secrets with short-term ones, significantly reducing the risk of compromise. Since the credentials are no longer stored with the application, rotating credentials no longer requires updating your applications and deploying changes to application clients.

For other types of secrets you might have in your organization:

- AWS credentials – We recommend AWS Identity and Access Management.
- Encryption keys – We recommend AWS Key Management Service.
- SSH keys – We recommend Amazon EC2 Instance Connect.
- Private keys and certificates – We recommend AWS Certificate Manager.

Get started with Secrets Manager

If you are new to Secrets Manager, start with one of the following tutorials:

- the section called “Replace hardcoded secrets ”
- the section called “Replace hardcoded DB credentials ”
- the section called “Alternating users rotation”
- the section called “Single user rotation”

Other tasks you can do with secrets:

- Manage secrets

- Control access to your secrets
- Get secrets
- Rotate secrets
- Monitor secrets
- Monitor secrets for compliance
- Create secrets in AWS CloudFormation

Compliance with standards

AWS Secrets Manager has undergone auditing for the multiple standards and can be part of your solution when you need to obtain compliance certification. For more information, see *Compliance validation*.

Pricing

When you use Secrets Manager, you pay only for what you use, with no minimum or setup fees. There is no charge for secrets that are marked for deletion. For the current complete pricing list, see AWS Secrets Manager Pricing. To monitor your costs, see the section called “Monitor Secrets Manager costs”.

You can use the AWS managed key `aws/secretsmanager` that Secrets Manager creates to encrypt your secrets for free. If you create your own KMS keys to encrypt your secrets, AWS charges you at the current AWS KMS rate. For more information, see AWS Key Management Service Pricing.

When you turn on automatic rotation (except managed rotation), Secrets Manager uses an AWS Lambda function to rotate the secret, and you are charged for the rotation function at the current Lambda rate. For more information, see AWS Lambda Pricing.

If you enable AWS CloudTrail on your account, you can obtain logs of the API calls that Secrets Manager sends out. Secrets Manager logs all events as management events. AWS CloudTrail stores the first copy of all management events for free. However, you can incur charges for Amazon S3 for log storage and for Amazon SNS if you enable notification. Also, if you set up additional trails, the additional copies of management events can incur costs. For more information, see AWS CloudTrail pricing.

You can use cost allocation tags in Secrets Manager to track and categorize expenses associated with specific secrets or projects. For more information, see the section called “Tag secrets” in this guide and Using AWS cost allocation tags in the AWS Billing User Guide.

Access AWS Secrets Manager

You can work with Secrets Manager in any of the following ways:

- Secrets Manager console
- Command line tools
- AWS SDKs
- HTTPS Query API
- AWS Secrets Manager endpoints

Secrets Manager console

You can manage your secrets using the browser-based Secrets Manager console and perform almost any task related to your secrets by using the console.

Command line tools

The AWS command line tools allows you to issue commands at your system command line to perform Secrets Manager and other AWS tasks. This can be faster and more convenient than using the console. The command line tools can be useful if you want to build scripts to perform AWS tasks.

When you enter commands in a command shell, there is a risk of the command history being accessed or utilities having access to your command parameters. See the section called “Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets”.

The command line tools automatically use the default endpoint for the service in an AWS Region. You can specify a different endpoint for your API requests. See the section called “Secrets Manager endpoints”.

AWS provides two sets of command line tools:

- AWS Command Line Interface (AWS CLI)
- AWS Tools for Windows PowerShell

AWS SDKs

The AWS SDKs consist of libraries and sample code for various programming languages and platforms. The SDKs include tasks such as cryptographically signing requests, managing errors, and retrying requests automatically. To download and install any of the SDKs, see Tools for Amazon Web Services.

The AWS SDKs automatically use the default endpoint for the service in an AWS Region. You can specify a different endpoint for your API requests. See the section called "Secrets Manager endpoints".

For SDK documentation, see:

- C++
- Go
- Java
- JavaScript
- Kotlin
- .NET
- PHP
- Python (Boto3)
- Ruby
- Rust
- SAP ABAP
- Swift

HTTPS Query API

The HTTPS Query API gives you programmatic access to Secrets Manager and AWS. The HTTPS Query API allows you to issue HTTPS requests directly to the service.

Although you can make direct calls to the Secrets Manager HTTPS Query API, we recommend that you use one of the SDKs instead. The SDK performs many useful tasks you otherwise must perform manually. For example, the SDKs automatically sign your requests and convert responses into a structure syntactically appropriate to your language.

To make HTTPS calls to Secrets Manager, you connect to ???.

AWS Secrets Manager endpoints

To connect programmatically to Secrets Manager, you use an *endpoint*, the URL of the entry point for the service. Secrets Manager endpoints are dual-stack endpoints, which means they support both IPv4 and IPv6.

Secrets Manager offers endpoints that support Federal Information Processing Standard (FIPS) 140-2 in some Regions.

Secrets Manager supports TLS 1.2 and 1.3. Secrets Manager supports PQTLS in all regions except China Regions.

> **Note**
>
> The Python AWS SDK and the AWS CLI attempt to call IPv6 and then IPv4 in sequence, so if you don't have IPv6 enabled, it can take some time before the call times out and retries with IPv4. To work around this issue, you can disable IPv6 completely or migrate to IPv6.

The following are the service endpoints for Secrets Manager. Note that the naming differs from the typical dual-stack naming convention. For information about using dual-stack addressing in Secrets Manager, see IPv4 and IPv6 access.

Region Name	Region	Endpoint	Protocol
US East (Ohio)	us-east-2	secretsmanager.us-east-2.amazonaws.com secretsmanager-fips.us-east-2.amazonaws.com	HTTPS HTTPS
US East (N. Virginia)	us-east-1	secretsmanager.us-east-1.amazonaws.com secretsmanager-fips.us-east-1.amazonaws.com	HTTPS HTTPS
US West (N.	us-west-1	secretsmanager.us-west-1.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
California)		secretsmanager-fips.us-west-1.amazonaws.com	HTTPS
US West (Oregon)	us-west-2	secretsmanager.us-west-2.amazonaws.com secretsmanager-fips.us-west-2.amazonaws.com	HTTPS HTTPS
Africa (Cape Town)	af-south-1	secretsmanager.af-south-1.amazonaws.com	HTTPS
Asia Pacific (Hong Kong)	ap-east-1	secretsmanager.ap-east-1.amazonaws.com	HTTPS
Asia Pacific (Hyderabad)	ap-south-2	secretsmanager.ap-south-2.amazonaws.com	HTTPS
Asia Pacific (Jakarta)	ap-southeast-3	secretsmanager.ap-southeast-3.amazonaws.com	HTTPS
Asia Pacific (Malaysia)	ap-southeast-5	secretsmanager.ap-southeast-5.amazonaws.com	HTTPS
Asia Pacific (Melbourne)	ap-southeast-4	secretsmanager.ap-southeast-4.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
Asia Pacific (Mumbai)	ap-south-1	secretsmanager.ap-south-1.amazonaws.com	HTTPS
Asia Pacific (New Zealand)	ap-southeast-6	secretsmanager.ap-southeast-6.amazonaws.com	HTTPS
Asia Pacific (Osaka)	ap-northeast-3	secretsmanager.ap-northeast-3.amazonaws.com	HTTPS
Asia Pacific (Seoul)	ap-northeast-2	secretsmanager.ap-northeast-2.amazonaws.com	HTTPS
Asia Pacific (Singapore)	ap-southeast-1	secretsmanager.ap-southeast-1.amazonaws.com	HTTPS
Asia Pacific (Sydney)	ap-southeast-2	secretsmanager.ap-southeast-2.amazonaws.com	HTTPS
Asia Pacific (Taipei)	ap-east-2	secretsmanager.ap-east-2.amazonaws.com	HTTPS
Asia Pacific (Thailand)	ap-southeast-7	secretsmanager.ap-southeast-7.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
Asia Pacific (Tokyo)	ap-northeast-1	secretsmanager.ap-northeast-1.amazonaws.com	HTTPS
Canada (Central)	ca-central-1	secretsmanager.ca-central-1.amazonaws.com secretsmanager-fips.ca-central-1.amazonaws.com	HTTPS HTTPS
Canada West (Calgary)	ca-west-1	secretsmanager.ca-west-1.amazonaws.com secretsmanager-fips.ca-west-1.amazonaws.com	HTTPS HTTPS
Europe (Frankfurt)	eu-central-1	secretsmanager.eu-central-1.amazonaws.com	HTTPS
Europe (Ireland)	eu-west-1	secretsmanager.eu-west-1.amazonaws.com	HTTPS
Europe (London)	eu-west-2	secretsmanager.eu-west-2.amazonaws.com	HTTPS
Europe (Milan)	eu-south-1	secretsmanager.eu-south-1.amazonaws.com	HTTPS
Europe (Paris)	eu-west-3	secretsmanager.eu-west-3.amazonaws.com	HTTPS
Europe (Spain)	eu-south-2	secretsmanager.eu-south-2.amazonaws.com	HTTPS
Europe (Stockholm)	eu-north-1	secretsmanager.eu-north-1.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
Europe (Zurich)	eu-central-2	secretsmanager.eu-central-2.amazonaws.com	HTTPS
Israel (Tel Aviv)	il-central-1	secretsmanager.il-central-1.amazonaws.com	HTTPS
Mexico (Central)	mx-central-1	secretsmanager.mx-central-1.amazonaws.com	HTTPS
Middle East (Bahrain)	me-south-1	secretsmanager.me-south-1.amazonaws.com	HTTPS
Middle East (UAE)	me-central-1	secretsmanager.me-central-1.amazonaws.com	HTTPS
South America (São Paulo)	sa-east-1	secretsmanager.sa-east-1.amazonaws.com	HTTPS
AWS GovCloud (US-East)	us-gov-east-1	secretsmanager.us-gov-east-1.amazonaws.com secretsmanager-fips.us-gov-east-1.amazonaws.com	HTTPS HTTPS
AWS GovCloud (US-West)	us-gov-west-1	secretsmanager.us-gov-west-1.amazonaws.com secretsmanager-fips.us-gov-west-1.amazonaws.com	HTTPS HTTPS

AWS Secrets Manager best practices

Secrets Manager provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Consider the following best practices for storing and managing secrets:

- Store credentials and other sensitive information in AWS Secrets Manager
- Find unprotected secrets in your code
- Choose an encryption key for your secret
- Use caching to retrieve secrets
- Rotate your secrets
- Mitigate risks of using CLI
- Limit access to secrets
- Replicate secrets
- Monitor secrets
- Run your infrastructure on private networks

Store credentials and other sensitive information in AWS Secrets Manager

Secrets Manager can help improve your security posture and compliance, and reduce the risk of unauthorized access to your sensitive information. Secrets Manager encrypts secrets at rest using encryption keys that you own and store in AWS Key Management Service (AWS KMS). When you retrieve a secret, Secrets Manager decrypts the secret and transmits it securely over TLS to your local environment. For more information, see *Create secrets*.

Find unprotected secrets in your code

CodeGuru Reviewer integrates with Secrets Manager to use a secrets detector that finds unprotected secrets in your code. The secrets detector searches for hardcoded passwords, database connection strings, user names, and more. For more information, see the section called “Amazon CodeGuru Reviewer”.

Amazon Q can scan your codebase for security vulnerabilities and code quality issues to improve the posture of your applications throughout the development cycle. For more information, see Scanning your code with Amazon Q in the *Amazon Q Developer User Guide*.

Choose an encryption key for your secret

For most cases, we recommend using the `aws/secretsmanager` AWS managed key to encrypt secrets. There is no cost for using it.

To be able to access a secret from another account or to apply a key policy to the encryption key, use a customer managed key to encrypt the secret.

- In the key policy, assign the value `secretsmanager.<region>.amazonaws.com` to the `kms:ViaService` condition key. This limits use of the key to only requests from Secrets Manager.
- To further limit use of the key to only requests from Secrets Manager with the correct context, use keys or values in the Secrets Manager encryption context as a condition for using the KMS key by creating:
  - A string condition operator in an IAM policy or key policy
  - A grant constraint in a grant

For more information, see the section called “Secret encryption and decryption”.

Use caching to retrieve secrets

To use your secrets most efficiently, we recommend you use one of the following supported Secrets Manager caching components to cache your secrets and update them only when required:

- Java with client-side caching
- Python with client-side caching
- .NET with client-side caching
- Go with client-side caching
- Rust with client-side caching

- AWS Parameters and Secrets Lambda Extension
- the section called "Amazon EKS"
- Use the section called "Secrets Manager Agent" to standardize consumption of secrets from Secrets Manager across environments such as AWS Lambda, Amazon Elastic Container Service, Amazon Elastic Kubernetes Service, and Amazon Elastic Compute Cloud.

Rotate your secrets

If you don't change your secrets for a long period of time, the secrets become more likely to be compromised. With Secrets Manager, you can set up automatic rotation as often as every four hours. Secrets Manager offers two strategies for rotation: Single user and Alternating users. For more information, see *Rotate secrets*.

Mitigate risks of using CLI

When you use the AWS CLI to invoke AWS operations, you enter those commands in a command shell. Most command shells offer features that could compromise your secrets, such as logging and the ability to see the last entered command. Before you use the AWS CLI to enter sensitive information, be sure to the section called "Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets".

Limit access to secrets

In IAM policy statements that control access to your secrets, use the principle of least privileged access. You can use IAM roles and policies, resource policies, and attribute-based access control (ABAC). For more information, see the section called "Authentication and access control".

Topics

- Block broad access to secrets
- Use caution with IP address conditions in policies
- Limit requests with VPC endpoint conditions

Block broad access to secrets

In identity policies that allow the action `PutResourcePolicy`, we recommend you use `BlockPublicPolicy: true`. This condition means that users can only attach a resource policy to a secret if the policy doesn't allow broad access.

Secrets Manager uses Zelkova automated reasoning to analyze resource policies for broad access. For more information about Zelkova, see How AWS uses automated reasoning to help you achieve security at scale on the AWS Security Blog.

The following example shows how to use `BlockPublicPolicy`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "secretsmanager:PutResourcePolicy",
        "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName-AbCdEf",
    "Condition": {
            "Bool": {
                "secretsmanager:BlockPublicPolicy": "true"
            }
        }
    }
}
```

Use caution with IP address conditions in policies

Use caution when you specify the IP address condition operators or the `aws:SourceIp` condition key in a policy statement that allows or denies access to Secrets Manager. For example, if you attach a policy that restricts AWS actions to requests from your corporate network IP address range to a secret, then your requests as an IAM user invoking the request from the corporate network work as expected. However, if you enable other services to access the secret on your behalf, such as when you enable rotation with a Lambda function, that function calls the Secrets Manager operations from an AWS-internal address space. Requests impacted by the policy with the IP address filter fail.

Also, the `aws:sourceIP` condition key is less effective when the request comes from an Amazon VPC endpoint. To restrict requests to a specific VPC endpoint, use the section called “Limit requests with VPC endpoint conditions”.

Limit requests with VPC endpoint conditions

To allow or deny access to requests from a particular VPC or VPC endpoint, use `aws:SourceVpc` to limit access to requests from the specified VPC or `aws:SourceVpce` to limit access to requests from the specified VPC endpoint. See the section called “Example: Permissions and VPCs”.

- `aws:SourceVpc` limits access to requests from the specified VPC.
- `aws:SourceVpce` limits access to requests from the specified VPC endpoint.

If you use these condition keys in a resource policy statement that allows or denies access to Secrets Manager secrets, you can inadvertently deny access to services that use Secrets Manager to access secrets on your behalf. Only some AWS services can run with an endpoint within your VPC. If you restrict requests for a secret to a VPC or VPC endpoint, then calls to Secrets Manager from a service not configured for the service can fail.

See the section called “VPC endpoints (AWS PrivateLink)”.

Replicate secrets

Secrets Manager can automatically replicate your secrets to multiple AWS Regions to meet your resiliency or disaster recovery requirements. For more information, see *Multi-region replication*.

Monitor secrets

Secrets Manager enables you to audit and monitor secrets through integration with AWS logging, monitoring, and notification services. For more information, see:

- the section called “Log with AWS CloudTrail ”
- the section called “Monitor with CloudWatch”
- the section called “Monitor secrets for compliance”
- the section called “Monitor Secrets Manager costs”
- the section called “Detect threats with GuardDuty”

Run your infrastructure on private networks

We recommend that you run as much of your infrastructure as possible on private networks that are not accessible from the public internet. You can establish a private connection between your VPC and Secrets Manager by creating an *interface VPC endpoint*. For more information, see [the section called “VPC endpoints (AWS PrivateLink)”](#).

AWS Secrets Manager tutorials

Topics

- Find unprotected secrets in your code with Amazon CodeGuru Reviewer
- Move hardcoded secrets to AWS Secrets Manager
- Move hardcoded database credentials to AWS Secrets Manager
- Set up alternating users rotation for AWS Secrets Manager
- Set up single user rotation for AWS Secrets Manager

Find unprotected secrets in your code with Amazon CodeGuru Reviewer

Amazon CodeGuru Reviewer is a service that uses program analysis and machine learning to detect potential defects that are difficult for developers to find and offers suggestions for improving your Java and Python code. CodeGuru Reviewer integrates with Secrets Manager to find unprotected secrets in your code. For the types of secrets it can find, see Types of secrets detected by CodeGuru Reviewer in the *Amazon CodeGuru Reviewer User Guide*.

Once you've found hardcoded secrets, take action to replace them:

- the section called “Replace hardcoded DB credentials ”
- the section called “Replace hardcoded secrets ”

Move hardcoded secrets to AWS Secrets Manager

If you have plaintext secrets in your code, we recommend that you rotate them and store them in Secrets Manager. Moving the secret to Secrets Manager solves the problem of the secret being visible to anyone who sees the code, because going forward, your code retrieves the secret directly from Secrets Manager. Rotating the secret revokes the current hardcoded secret so that it is no longer valid.

For database credential secrets, see Move hardcoded database credentials to AWS Secrets Manager.

Before you begin, you need to determine who needs access to the secret. We recommend using two IAM roles to manage permission to your secret:

- A role that manages the secrets in your organization. For more information, see the section called "Secrets Manager administrator permissions". You'll create and rotate the secret using this role.
- A role that can use the secret at runtime, for example in this tutorial you use *RoleToRetrieveSecretAtRuntime*. Your code assumes this role to retrieve the secret. In this tutorial, you grant the role only the permission to retrieve one secret value, and you grant permission by using the secret's resource policy. For other alternatives, see the section called "Next steps".

Steps:

- Step 1: Create the secret
- Step 2: Update your code
- Step 3: Update the secret
- Next steps

Step 1: Create the secret

The first step is to copy the existing hardcoded secret into Secrets Manager. If the secret is related to an AWS resource, store it in the same Region as the resource. Otherwise, store it in the Region that has the lowest latency for your use case.

To create a secret (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following:

   a. For **Secret type**, choose **Other type of secret**.

   b. Enter your secret as **Key/value pairs** or in **Plaintext**. Some examples:

      API key

         Enter as key/value pairs:

         `ClientID` : *`my_client_id`*

         `ClientSecret` : *`wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`*

OAuth token

Enter as plaintext:

AKIAI44QH8DHBEXAMPLE

Digital certificate

Enter as plaintext:

```
-----BEGIN CERTIFICATE-----
EXAMPLE
-----END CERTIFICATE-----
```

Private key

Enter as plaintext:

```
---- BEGIN PRIVATE KEY ----
EXAMPLE
---- END PRIVATE KEY ----
```

c. For **Encryption key**, choose **aws/secretsmanager** to use the AWS managed key for Secrets Manager. There is no cost for using this key. You can also use your own customer managed key, for example to access the secret from another AWS account. For information about the costs of using a customer managed key, see Pricing.

d. Choose **Next**.

4. On the **Choose secret type** page, do the following:

a. Enter a descriptive **Secret name** and **Description**.

b. In **Resource permissions**, choose **Edit permissions**. Paste the following policy, which allows *RoleToRetrieveSecretAtRuntime* to retrieve the secret, and then choose **Save**.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
                "Principal": {
                    "AWS":
 "arn:aws:iam::111122223333:role/RoleToRetrieveSecretAtRuntime"
                },
                "Action": "secretsmanager:GetSecretValue",
                "Resource": "*"
            }
        ]
    }
```

c. At the bottom of the page, choose **Next**.

5. On the **Configure rotation** page, keep rotation off. Choose **Next**.
6. On the **Review** page, review your secret details, and then choose **Store**.

Step 2: Update your code

Your code must assume the IAM role *RoleToRetrieveSecretAtRuntime* to be able to retrieve the secret. For more information, see Switching to an IAM role (AWS API).

Next, you update your code to retrieve the secret from Secrets Manager using the sample code provided by Secrets Manager.

To find the sample code

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. On the **Secrets** page, choose your secret.
3. Scroll down to **Sample code**. Choose your programming language, and then copy the code snippet.

In your application, remove the hardcoded secret and paste the code snippet. Depending on your code language, you might need to add a call to the function or method in the snippet.

Test that your application works as expected with the secret in place of the hardcoded secret.

Step 3: Update the secret

The last step is to revoke and update the hardcoded secret. Refer to the source of the secret to find instructions to revoke and update the secret. For example, you might need to deactivate the current secret and generate a new secret.

To update the secret with the new value

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Secrets**, and then choose the secret.
3. On the **Secret details** page, scroll down and choose **Retrieve secret value**, and then choose **Edit**.
4. Update the secret and then choose **Save**.

Next, test that your application works as expected with the new secret.

Next steps

After you remove a hardcoded secret from your code, some ideas to consider next:

- To find hardcoded secrets in your Java and Python applications, we recommend Amazon CodeGuru Reviewer.
- You can improve performance and reduce costs by caching secrets. For more information, see *Get secrets*.
- For secrets that you access from multiple Regions, consider replicating your secret to improve latency. For more information, see *Multi-region replication*.
- In this tutorial, you granted `RoleToRetrieveSecretAtRuntime` only the permission to retrieve the secret value. To grant the role more permissions, for example to get metadata about the secret or to view a list of secrets, see the section called “Resource-based policies”.
- In this tutorial, you granted permission to `RoleToRetrieveSecretAtRuntime` by using the secret's resource policy. For other ways to grant permission, see the section called “Identity-based policies”.

Move hardcoded database credentials to AWS Secrets Manager

If you have plaintext database credentials in your code, we recommend that you move the credentials to Secrets Manager and then rotate them immediately. Moving the credentials to Secrets Manager solves the problem of the credentials being visible to anyone who sees the code, because going forward, your code retrieves the credentials directly from Secrets Manager. Rotating the secret updates the password and then revokes the current hardcoded password so that it is no longer valid.

For Amazon RDS, Amazon Redshift, and Amazon DocumentDB databases, use the steps in this page to move hardcoded credentials to Secrets Manager. For other types of credentials and other secrets, see the section called "Replace hardcoded secrets ".

Before you begin, you need to determine who needs access to the secret. We recommend using two IAM roles to manage permission to your secret:

- A role that manages the secrets in your organization. For more information, see the section called "Secrets Manager administrator permissions". You'll create and rotate the secret using this role.
- A role that can use the credentials at runtime, *RoleToRetrieveSecretAtRuntime* in this tutorial. Your code assumes this role to retrieve the secret.

Steps:

Step 1: Create the secret

The first step is to copy the existing hardcoded credentials into a secret in Secrets Manager. For the lowest latency, store the secret in the same Region as the database.

To create a secret

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following:

   a. For **Secret type**, choose the type of database credentials to store:

      - **Amazon RDS database**
      - **Amazon DocumentDB database**
      - **Amazon Redshift data warehouse**.
      - For other types of secrets, see Replace hardcoded secrets .

   b. For **Credentials**, enter the existing hardcoded credentials for the database.
   c. For **Encryption key**, choose **aws/secretsmanager** to use the AWS managed key for Secrets Manager. There is no cost for using this key. You can also use your own customer managed key, for example to access the secret from another AWS account. For information about the costs of using a customer managed key, see Pricing.
   d. For **Database**, choose your database.
   e. Choose **Next**.
4. On the **Configure secret** page, do the following:
   a. Enter a descriptive **Secret name** and **Description**.
   b. In **Resource permissions**, choose **Edit permissions**. Paste the following policy, which allows *RoleToRetrieveSecretAtRuntime* to retrieve the secret, and then choose **Save**.

      JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS":
 "arn:aws:iam::111122223333:role/RoleToRetrieveSecretAtRuntime"
            },
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "*"
        }
    ]
}
```

   c. At the bottom of the page, choose **Next**.
5. On the **Configure rotation** page, keep rotation off for now. You'll turn it on later. Choose **Next**.
6. On the **Review** page, review your secret details, and then choose **Store**.

Step 2: Update your code

Your code must assume the IAM role *`RoleToRetrieveSecretAtRuntime`* to be able to retrieve the secret. For more information, see Switching to an IAM role (AWS API).

Next, you update your code to retrieve the secret from Secrets Manager using the sample code provided by Secrets Manager.

To find the sample code

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. On the **Secrets** page, choose your secret.
3. Scroll down to **Sample code**. Choose your language, and then copy the code snippet.

In your application, remove the hardcoded credentials and paste the code snippet. Depending on your code language, you might need to add a call to the function or method in the snippet.

Test that your application works as expected with the secret in place of the hardcoded credentials.

Step 3: Rotate the secret

The last step is to revoke the hardcoded credentials by rotating the secret. *Rotation* is the process of periodically updating a secret. When you rotate a secret, you update the credentials in both the secret and the database. Secrets Manager can automatically rotate a secret for you on a schedule you set.

Part of setting up rotation is ensuring that the Lambda rotation function can access both Secrets Manager and your database. When you turn on automatic rotation, Secrets Manager creates the Lambda rotation function in the same VPC as your database so that it has network access to the database. The Lambda rotation function must also be able to make calls to Secrets Manager to update the secret. We recommend that you create a Secrets Manager endpoint in the VPC so that calls from Lambda to Secrets Manager don't leave AWS infrastructure. For instructions, see the section called “VPC endpoints (AWS PrivateLink)”.

To turn on rotation

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. On the **Secrets** page, choose your secret.

3. On the **Secret details** page, in the **Rotation configuration** section, choose **Edit rotation**.
4. In the **Edit rotation configuration** dialog box, do the following:

   a. Turn on **Automatic rotation**.

   b. Under **Rotation schedule**, enter your schedule in UTC time zone.

   c. Choose **Rotate immediately when the secret is stored** to rotate your secret when you save your changes.

   d. Under **Rotation function**, choose **Create a new Lambda function** and enter a name for your new function. Secrets Manager adds "SecretsManager" to the beginning of your function name.

   e. For **Rotation strategy**, choose **Single user**.

   f. Choose **Save**.

To check that the secret rotated

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Secrets**, and then choose the secret.
3. On the **Secret details** page, scroll down and choose **Retrieve secret value**.

   If the secret value changed, then rotation succeeded. If the secret value didn't change, you need to Troubleshoot rotation by looking at the CloudWatch Logs for the rotation function.

Test that your application works as expected with the rotated secret.

Next steps

After you remove a hardcoded secret from your code, some ideas to consider next:

- You can improve performance and reduce costs by caching secrets. For more information, see *Get secrets*.
- You can choose a different rotation schedule. For more information, see the section called “Rotation schedules”.
- To find hardcoded secrets in your Java and Python applications, we recommend Amazon CodeGuru Reviewer.

Set up alternating users rotation for AWS Secrets Manager

In this tutorial, you learn how to set up alternating users rotation for a secret that contains database credentials. *Alternating users rotation* is a rotation strategy where Secrets Manager clones the user and then alternates which user's credentials are updated. This strategy is a good choice if you need high availability for your secret, because one of the alternating users has current credentials to the database while the other one is being updated. For more information, see the section called "Alternating users".

To set up alternating users rotation, you need two secrets:

- One secret with the credentials that you want to rotate.
- A second secret that has admin credentials.

  This user has permissions to clone the first user and change the first users' password. In this tutorial, you have Amazon RDS create this secret for an admin user. Amazon RDS also manages the admin password rotation. For more information, see the section called "Managed rotation".

The first part of this tutorial is setting up a realistic environment. To show you how rotation works, this tutorial uses an example Amazon RDS MySQL database. For security, the database is in a VPC that restricts inbound internet access. To connect to the database from your local computer through the internet, you use a *bastion host*, a server in the VPC that can connect to the database, but that also allows SSH connections from the internet. The bastion host in this tutorial is an Amazon EC2 instance, and the security groups for the instance prevent other types of connections.

After you finish the tutorial, we recommend that you clean up the resources from the tutorial. Don't use them in a production setting.

Secrets Manager rotation uses an AWS Lambda function to update the secret and the database. For information about the costs of using a Lambda function, see Pricing.

Tutorial:

- Step 4: Clean up resources
- Next steps

Permissions

For the tutorial prerequisites, you need administrative permissions to your AWS account. In a production setting, it is a best practice to use different roles for each of the steps. For example, a role with database admin permissions would create the Amazon RDS database, and a role with network admin permissions would set up the VPC and security groups. For the tutorial steps, we recommend you continue using the same identity.

For information about how to set up permissions in a production environment, see the section called "Authentication and access control".

Prerequisites

For this tutorial, you need the following:

- Prereq A: Amazon VPC
- Prereq B: Amazon EC2 instance
- Prereq C: Amazon RDS database and a Secrets Manager secret for the admin credentials
- Prereq D: Allow your local computer to connect to the EC2 instance

Prereq A: Amazon VPC

In this step, you create a VPC that you can launch an Amazon RDS database and an Amazon EC2 instance into. In a later step, you'll use your computer to connect through the internet to the bastion and then to the database, so you need to allow traffic out of the VPC. To do this, Amazon VPC attaches an internet gateway to the VPC and adds a route in the route table so that traffic destined for outside the VPC is sent to the internet gateway.

Within the VPC, you create a Secrets Manager endpoint and an Amazon RDS endpoint. When you set up automatic rotation in a later step, Secrets Manager creates a Lambda rotation function within the VPC so that it can access the database. The Lambda rotation function also calls Secrets Manager to update the secret, and it calls Amazon RDS to get the database connection information. By creating endpoints within the VPC, you ensure that calls from the Lambda function to Secrets Manager and Amazon RDS don't leave AWS infrastructure. Instead, they are routed to the endpoints within the VPC.

To create a VPC

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Choose **Create VPC**.
3. On the **Create VPC** page, choose **VPC and more**.
4. Under **Name tag auto-generation**, under **Auto-generate**, enter **`SecretsManagerTutorial`**.
5. For **DNS options**, choose both **`Enable DNS hostnames`** and **`Enable DNS resolution`**.
6. Choose **Create VPC**.

To create a Secrets Manager endpoint within the VPC

1. In the Amazon VPC console, under **Endpoints**, choose **Create Endpoint**.
2. Under **Endpoint settings**, for **Name**, enter **`SecretsManagerTutorialEndpoint`**.
3. Under **Services**, enter **`secretsmanager`** to filter the list, and then select the Secrets Manager endpoint in your AWS Region. For example, in the US East (N. Virginia), choose `com.amazonaws.us-east-1.secretsmanager`.
4. For **VPC**, choose **`vpc**** (SecretsManagerTutorial)`**.
5. For **Subnets**, select all **Availability Zones**, and then for each one, choose a **Subnet ID** to include.
6. For **IP address type**, choose **`IPv4`**.
7. For **Security groups**, choose the default security group.
8. For **Policy**, choose **`Full access`**.
9. Choose **Create endpoint**.

To create an Amazon RDS endpoint within the VPC

1. In the Amazon VPC console, under **Endpoints**, choose **Create Endpoint**.
2. Under **Endpoint settings**, for **Name**, enter **`RDSTutorialEndpoint`**.
3. Under **Services**, enter **`rds`** to filter the list, and then select the Amazon RDS endpoint in your AWS Region. For example, in the US East (N. Virginia), choose `com.amazonaws.us-east-1.rds`.
4. For **VPC**, choose **`vpc**** (SecretsManagerTutorial)`**.

5. For **Subnets**, select all **Availability Zones**, and then for each one, choose a **Subnet ID** to include.
6. For **IP address type**, choose **`IPv4`**.
7. For **Security groups**, choose the default security group.
8. For **Policy**, choose **`Full access`**.
9. Choose **Create endpoint**.

Prereq B: Amazon EC2 instance

The Amazon RDS database you create in a later step will be in the VPC, so to access it, you need a bastion host. The bastion host is also in the VPC, but in a later step, you configure a security group to allow your local computer to connect to the bastion host with SSH.

To create an EC2 instance for a bastion host

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Choose **Instances** and then choose **Launch Instances**.
3. Under **Name and tags**, for **Name**, enter **`SecretsManagerTutorialInstance`**.
4. Under **Application and OS Images**, keep the default **`Amazon Linux 2 AMI (HMV) Kernel 5.10`**.
5. Under **Instance type**, keep the default **`t2.micro`**.
6. Under **Key pair**, choose **Create key pair**.

   In the **Create key pair** dialog box, for **Key pair name**, enter **`SecretsManagerTutorialKeyPair`**, and then choose **Create key pair**.

   The key pair is automatically downloaded.
7. Under **Network settings**, choose **Edit**, and then do the following:

   a. For **VPC**, choose **`vpc-**** SecretsManagerTutorial`**.

   b. For **Auto-assign Public IP**, choose **`Enable`**.

   c. For **Firewall**, choose **Select existing security group**.

   d. For **Common security groups**, choose **`default`**.
8. Choose **Launch instance**.

Prereq C: Amazon RDS database and a Secrets Manager secret for the admin credentials

In this step, you create an Amazon RDS MySQL database and configure it so that Amazon RDS creates a secret to contain the admin credentials. Then Amazon RDS automatically manages rotation of the admin secret for you. For more information, see Managed rotation.

As part of creating your database, you specify the bastion host you created in the previous step. Then Amazon RDS sets up security groups so that the database and the instance can access each other. You add a rule to the security group attached to the instance to allow your local computer to connect to it as well.

To create an Amazon RDS database with an Secrets Manager secret that contains the admin credentials

1. In the Amazon RDS console, choose **Create database**.
2. In the **Engine options** section, for **Engine type**, choose **`MySQL`**.
3. In the **Templates** section, choose **`Free tier`**.
4. In the **Settings** section, do the following:

    a. For **DB instance identifier**, enter **`SecretsManagerTutorial`**.

    b. Under **Credential settings**, select **Manage master credentials in AWS Secrets Manager**.
5. In the **Connectivity** section, for **Computer resource**, choose **Connect to an EC2 computer resource**, and then for **EC2 Instance**, choose **`SecretsManagerTutorialInstance`**.
6. Choose **Create database**.

Prereq D: Allow your local computer to connect to the EC2 instance

In this step, you configure the EC2 instance you created in Prereq B to allow your local computer to connect to it. To do this, you edit the security group that Amazon RDS added in Prereq C to include a rule that allows your computer's IP address to connect with SSH. The rule allows your local computer (identified by your current IP address) to connect to the bastion host by using SSH over the internet.

To allow your local computer to connect to the EC2 instance

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.

2. On the EC2 instance **SecretsManagerTutorialInstance**, on the **Security** tab, under **Security groups**, choose **`sg-*** (ec2-rds-X)`**.
3. Under **Input rules**, choose **Edit inbound rules**.
4. Choose **Add rule**, and then for the rule, do the following:

    a. For **Type**, choose **`SSH`**.

    b. For **Source type**, choose **`My IP`**.

Step 1: Create an Amazon RDS database user

First, you need a user whose credentials will be stored in the secret. To create the user, log into the Amazon RDS database with admin credentials. For simplicity, in the tutorial, you create a user with full permission to a database. In a production setting, this is not typical, and we recommend that you follow the principle of least privilege.

To connect to the database, you use a MySQL client tool. In this tutorial, you use MySQL Workbench, a GUI-based application. To install MySQL Workbench, see Download MySQL Workbench.

To connect to the database, create a connection configuration in MySQL Workbench. For the configuration, you need some information from both Amazon EC2 and Amazon RDS.

To create a database connection in MySQL Workbench

1. In MySQL Workbench, next to **MySQL Connections**, choose the (+) button.
2. In the **Setup New Connection** dialog box, do the following:

    a. For **Connection Name**, enter **`SecretsManagerTutorial`**.

    b. For **Connection Method**, choose **`Standard TCP/IP over SSH`**.

    c. On the **Parameters** tab, do the following:

        i. For **SSH Hostname**, enter the public IP address of the Amazon EC2 instance.

        You can find the IP address on the Amazon EC2 console by choosing the instance **SecretsManagerTutorialInstance**. Copy the IP address under **Public IPv4 DNS**.

        ii. For **SSH Username**, enter **`ec2-user`**.

        iii. For **SSH Keyfile**, choose the key pair file **SecretsManagerTutorialKeyPair.pem** you downloaded in the previous prerequisite.

   iv. For **MySQL Hostname**, enter the Amazon RDS endpoint address.

       You can find the endpoint address on the Amazon RDS console by choosing the database instance **secretsmanagertutorialdb**. Copy the address under **Endpoint**.

   v. For **Username**, enter **`admin`**.

d. Choose **OK**.

To retrieve the admin password

1. In the Amazon RDS console, navigate to your database.
2. On the **Configuration** tab, under **Master Credentials ARN**, choose **Manage in Secrets Manager**.

   The Secrets Manager console opens.
3. In the secret details page, choose **Retrieve secret value**.
4. The password appears in the **Secret value** section.

To create a database user

1. In MySQL Workbench, choose the connection **SecretsManagerTutorial**.
2. Enter the admin password you retrieved from the secret.
3. In MySQL Workbench, in the **Query** window, enter the following commands (including a strong password) and then choose **Execute**. The rotation function tests the updated secret by using SELECT, so the **`appuser`** must have that privilege at minimum.

```
CREATE DATABASE myDB;
CREATE USER 'appuser'@'%' IDENTIFIED BY 'EXAMPLE-PASSWORD';
GRANT SELECT ON myDB . * TO 'appuser'@'%';
```

   In the **Output** window, you see the commands are successful.

Step 2: Create a secret for the user credentials

Next, you create a secret to store the credentials of the user you just created. This is the secret you'll be rotating. You turn on automatic rotation, and to indicate the alternating users strategy, you choose a separate superuser secret that has permission to change the first user's password.

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following:
   a. For **Secret type**, choose **Credentials for Amazon RDS database**.
   b. For **Credentials**, enter the username **`appuser`** and the password you entered for the database user you created using MySQL Workbench.
   c. For **Database**, choose **secretsmanagertutorialdb**.
   d. Choose **Next**.
4. On the **Configure secret** page, for **Secret name**, enter **`SecretsManagerTutorialAppuser`** and then choose **Next**.
5. On the **Configure rotation** page, do the following:
   a. Turn on **Automatic rotation**.
   b. For **Rotation schedule**, set a schedule of **Days**: **2** Days with **Duration**: **2h**. Keep **Rotate immediately** selected.
   c. For **Rotation function**, choose **Create a rotation function**, and then for the function name, enter **`tutorial-alternating-users-rotation`**.
   d. For **Rotation strategy**, choose **Alternating users**, and then under **Admin credential secret**, choose the secret named **rds!cluster...** which has a **Description** that includes the name of the database you created in this tutorial **`secretsmanagertutorial`**, for example `Secret associated with primary RDS DB instance: arn:aws:rds:`*`Region`*`:`*`AccountId`*`:db:secretsmanagertutorial`.
   e. Choose **Next**.
6. On the **Review** page, choose **Store**.

   Secrets Manager returns to the secret details page. At the top of the page, you can see the rotation configuration status. Secrets Manager uses CloudFormation to create resources such as the Lambda rotation function and an execution role that runs the Lambda function. When CloudFormation finishes, the banner changes to **Secret scheduled for rotation**. The first rotation is complete.

Step 3: Test the rotated secret

Now that the secret is rotated, you can check that the secret contains valid new credentials. The password in the secret has changed from the original credentials.

To retrieve the new password from the secret

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Secrets**, and then choose the secret **`SecretsManagerTutorialAppuser`**.
3. On the **Secret details** page, scroll down and choose **Retrieve secret value**.
4. In the **Key/value** table, copy the **Secret value** for **`password`**.

To test the credentials

1. In MySQL Workbench, right-click the connection **SecretsManagerTutorial** and then choose **Edit Connection**.
2. In the **Manage Server Connections** dialog box, for **Username**, enter **`appuser`**, and then choose **Close**.
3. Back in MySQL Workbench, choose the connection **SecretsManagerTutorial**.
4. In the **Open SSH Connection** dialog box, for **Password**, paste the password you retrieved from the secret, and then choose **OK**.

   If the credentials are valid, then MySQL Workbench opens to the design page for the database.

This shows that the secret rotation is successful. The credentials in the secret have been updated and it is a valid password to connect to the database.

Step 4: Clean up resources

If you want to try another rotation strategy, *single user rotation*, skip cleaning up resources and go to the section called “Single user rotation”.

Otherwise, to avoid potential charges, and to remove the EC2 instance that has access to the internet, delete the following resources you created in this tutorial and its prerequisites:

- Amazon RDS database instance. For instructions, see Deleting a DB instance in the *Amazon RDS User Guide*.

- Amazon EC2 instance. For instructions, see Terminate an instance in the *Amazon EC2 User Guide*.
- Secrets Manager secret `SecretsManagerTutorialAppuser`. For instructions, see the section called "Delete a secret".
- Secrets Manager endpoint. For instructions, see Delete a VPC endpoint in the *AWS PrivateLink Guide*.
- VPC endpoint. For instructions, see Delete your VPC in the *AWS PrivateLink Guide*.

Next steps

- Learn how to retrieve secrets in your applications.
- Learn about other rotation schedules.

Set up single user rotation for AWS Secrets Manager

In this tutorial, you learn how to set up single user rotation for a secret that contains database credentials. *Single user rotation* is a rotation strategy where Secrets Manager updates a user's credentials in both the secret and the database. For more information, see the section called "Single user".

After you finish the tutorial, we recommend that you clean up the resources from the tutorial. Don't use them in a production setting.

Secrets Manager rotation uses an AWS Lambda function to update the secret and the database. For information about the costs of using a Lambda function, see Pricing.

Contents

Permissions

For the tutorial prerequisites, you need administrative permissions to your AWS account. In a production setting, it is a best practice to use different roles for each of the steps. For example, a role with database admin permissions would create the Amazon RDS database, and a role with network admin permissions would set up the VPC and security groups. For the tutorial steps, we recommend you continue using the same identity.

For information about how to set up permissions in a production environment, see the section called "Authentication and access control".

Prerequisites

The prerequisite for this tutorial is the section called "Alternating users rotation". Don't clean up the resources at the end of the first tutorial. After that tutorial, you have a realistic environment with an Amazon RDS database and a Secrets Manager secret that contains admin credentials for the database. You also have a second secret that contains credentials for a database user, but you don't use that secret in this tutorial.

You also have a connection configured in MySQL Workbench to connect to the database with the admin credentials.

Step 1: Create an Amazon RDS database user

First, you need a user whose credentials will be stored in the secret. To create the user, log into the Amazon RDS database with admin credentials that are stored in a secret. For simplicity, in the tutorial, you create a user with full permission to a database. In a production setting, this is not typical, and we recommend that you follow the principle of least privilege.

To retrieve the admin password

1. In the Amazon RDS console, navigate to your database.
2. On the **Configuration** tab, under **Master Credentials ARN**, choose **Manage in Secrets Manager**.

   The Secrets Manager console opens.
3. In the secret details page, choose **Retrieve secret value**.
4. The password appears in the **Secret value** section.

To create a database user

1. In MySQL Workbench, right-click the connection **SecretsManagerTutorial** and then choose **Edit Connection**.
2. In the **Manage Server Connections** dialog box, for **Username**, enter **`admin`**, and then choose **Close**.
3. Back in MySQL Workbench, choose the connection **SecretsManagerTutorial**.
4. Enter the admin password you retrieved from the secret.
5. In MySQL Workbench, in the **Query** window, enter the following commands (including a strong password) and then choose **Execute**. The rotation function tests the updated secret by using SELECT, so the **`dbuser`** must have that privilege at minimum.

   ```
   CREATE USER 'dbuser'@'%' IDENTIFIED BY 'EXAMPLE-PASSWORD';
   GRANT SELECT ON myDB . * TO 'dbuser'@'%';
   ```

   In the **Output** window, you see the commands are successful.

Step 2: Create a secret for the database user credentials

Next, you create a secret to store the credentials of the user you just created, and you turn on automatic rotation, including an immediate rotation. Secrets Manager rotates the secret, which means the password is programmatically generated – no human has seen this new password. Having the rotation begin immediately can also help you determine if rotation is set up correctly.

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following:

   a. For **Secret type**, choose **Credentials for Amazon RDS database**.

   b. For **Credentials**, enter the username **`dbuser`** and the password you entered for the database user you created using MySQL Workbench.

   c. For **Database**, choose **secretsmanagertutorialdb**.

   d. Choose **Next**.
4. On the **Configure secret** page, for **Secret name**, enter **`SecretsManagerTutorialDbuser`** and then choose **Next**.

5. On the **Configure rotation** page, do the following:

    a. Turn on **Automatic rotation**.

    b. For **Rotation schedule**, set a schedule of **Days**: **2** Days with **Duration**: **2h**. Keep **Rotate immediately** selected.

    c. For **Rotation function**, choose **Create a rotation function**, and then for the function name, enter **`tutorial-single-user-rotation`**.

    d. For **Rotation strategy**, choose **Single user**.

    e. Choose **Next**.

6. On the **Review** page, choose **Store**.

    Secrets Manager returns to the secret details page. At the top of the page, you can see the rotation configuration status. Secrets Manager uses CloudFormation to create resources such as the Lambda rotation function and an execution role that runs the Lambda function. When CloudFormation finishes, the banner changes to **Secret scheduled for rotation**. The first rotation is complete.

Step 3: Test the rotated password

After the first secret rotation, which might take a few seconds, you can check that the secret still contains valid credentials. The password in the secret has changed from the original credentials.

To retrieve the new password from the secret

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Secrets**, and then choose the secret **`SecretsManagerTutorialDbuser`**.
3. On the **Secret details** page, scroll down and choose **Retrieve secret value**.
4. In the **Key/value** table, copy the **Secret value** for **`password`**.

To test the credentials

1. In MySQL Workbench, right-click the connection **SecretsManagerTutorial** and then choose **Edit Connection**.
2. In the **Manage Server Connections** dialog box, for **Username**, enter **`dbuser`**, and then choose **Close**.
3. Back in MySQL Workbench, choose the connection **SecretsManagerTutorial**.

4. In the **Open SSH Connection** dialog box, for **Password**, paste the password you retrieved from the secret, and then choose **OK**.

   If the credentials are valid, then MySQL Workbench opens to the design page for the database.

Step 4: Clean up resources

To avoid potential charges, delete the secret you created in this tutorial. For instructions, see the section called “Delete a secret”.

To clean up resources created in the previous tutorial, see the section called “Step 4: Clean up resources”.

Next steps

- Learn how to retrieve secrets in your applications. See *Get secrets*.
- Learn about other rotation schedules. See the section called “Rotation schedules”.

Create an AWS Secrets Manager secret

A *secret* can be a password, a set of credentials such as a user name and password, an OAuth token, or other secret information that you store in an encrypted form in Secrets Manager.

> **Tip**
>
> For Amazon RDS and Amazon Redshift admin user credentials, we recommend you use managed secrets. You create the managed secret through the managing service, and then you can use managed rotation.

When you use the console to store database credentials for a source database that is replicated to other Regions, the secret contains connection information for the source database. If you then replicate the secret, the replicas are copies of the source secret and contain the same connection information. You can add additional key/value pairs to the secret for regional connection information.

To create a secret, you need the permissions granted by the SecretsManagerReadWrite managed policy.

Secrets Manager generates a CloudTrail log entry when you create a secret. For more information, see the section called “Log with AWS CloudTrail ”.

To create a secret (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose **Store a new secret**.
3. On the **Choose secret type** page, do the following:
    a. For **Secret type**, do one of the following:
        - To store database credentials, choose the type of database credentials to store. Then choose the **Database** and then enter the **Credentials**.
        - To store API keys, access tokens, credentials that aren't for databases, choose **Other type of secret**.

In **Key/value pairs**, either enter your secret in JSON **Key/value** pairs, or choose the **Plaintext** tab and enter the secret in any format. You can store up to 65536 bytes in the secret. Some examples:

API key

Enter as key/value pairs:

`ClientID` : *`my_client_id`*

`ClientSecret` : *`wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`*

OAuth token

Enter as plaintext:

`AKIAI44QH8DHBEXAMPLE`

Digital certificate

Enter as plaintext:

```
-----BEGIN CERTIFICATE-----
EXAMPLE
-----END CERTIFICATE-----
```

Private key

Enter as plaintext:

```
---- BEGIN PRIVATE KEY ----
EXAMPLE
---- END PRIVATE KEY ----
```

- To store a managed external secrets from an Secrets Manager partner, choose **Partner secret**. Then choose the partner and provide the details that identify the secret for the partner. For details, see Using AWS Secrets Manager managed external secrets to manage Third Party secrets.

b. For **Encryption key**, choose the AWS KMS key that Secrets Manager uses to encrypt the secret value. For more information, see Secret encryption and decryption.

- For most cases, choose **aws/secretsmanager** to use the AWS managed key for Secrets Manager. There is no cost for using this key.
- If you need to access the secret from another AWS account, or if you want to use your own KMS key so that you can rotate it or apply a key policy to it, choose a customer managed key from the list or choose **Add new key** to create one. For information about the costs of using a customer managed key, see Pricing.

  You must have the section called “Permissions for the KMS key”. For information about cross-account access, see the section called “Cross-account access”.

c. Choose **Next**.

4. On the **Configure secret** page, do the following:

   a. Enter a descriptive **Secret name** and **Description**. Secret names can contain 1-512 alphanumeric and /_+=.@- characters.

   b. (Optional) If you are created an external secret, enter the metadata required by the Secrets Manager partner that holds the secret.

   c. (Optional) In the **Tags** section, add tags to your secret. For tagging strategies, see the section called “Tag secrets”. Don't store sensitive information in tags because they aren't encrypted.

   d. (Optional) In **Resource permissions**, to add a resource policy to your secret, choose **Edit permissions**. For more information, see the section called “Resource-based policies”.

   e. (Optional) In **Replicate secret**, to replicate your secret to another AWS Region, choose **Replicate secret**. You can replicate your secret now or come back and replicate it later. For more information, see Multi-region replication.

   f. Choose **Next**.

5. (Optional) On the **Configure rotation** page, you can turn on automatic rotation. You can also keep rotation off for now and then turn it on later. For more information, see Rotate secrets. Choose **Next**.

6. On the **Review** page, review your secret details, and then choose **Store**.

   Secrets Manager returns to the list of secrets. If your new secret doesn't appear, choose the refresh button.

AWS CLI

When you enter commands in a command shell, there is a risk of the command history being accessed or utilities having access to your command parameters. See the section called “Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets”.

Example Create a secret from database credentials in a JSON file

The following `create-secret` example creates a secret from credentials in a file. For more information, see Loading AWS CLI parameters from a file in the AWS CLI User Guide.

For Secrets Manager to be able to rotate the secret, you must make sure the JSON matches the JSON structure of a secret.

```
aws secretsmanager create-secret \
    --name MyTestSecret \
    --secret-string file://mycreds.json
```

Contents of mycreds.json:

```
{
    "engine": "mysql",
    "username": "saanvis",
    "password": "EXAMPLE-PASSWORD",
    "host": "my-database-endpoint.us-west-2.rds.amazonaws.com",
    "dbname": "myDatabase",
    "port": "3306"
}
```

Example Create a secret

The following `create-secret` example creates a secret with two key-value pairs.

```
aws secretsmanager create-secret \
    --name MyTestSecret \
    --description "My test secret created with the CLI." \
    --secret-string '{"user":"diegor","password":"EXAMPLE-PASSWORD"}'
```

Example Create a secret

The following `create-secret` example creates a secret with two tags.

```
aws secretsmanager create-secret \
    --name MyTestSecret \
    --description "My test secret created with the CLI." \
    --secret-string '{"user":"diegor","password":"EXAMPLE-PASSWORD"}'  \
    --tags '[{"Key": "FirstTag", "Value": "FirstValue"}, {"Key": "SecondTag", "Value":
 "SecondValue"}]'
```

AWS SDK

To create a secret by using one of the AWS SDKs, use the `CreateSecret` action. For more information, see the section called "AWS SDKs".

What's in a Secrets Manager secret?

In Secrets Manager, a *secret* consists of secret information, the *secret value*, plus metadata about the secret. A secret value can be a string or binary.

To store multiple string values in one secret, we recommend that you use a JSON text string with key-value pairs, for example:

```
{
  "host"       : "ProdServer-01.databases.example.com",
  "port"       : "8888",
  "username"   : "administrator",
  "password"   : "EXAMPLE-PASSWORD",
  "dbname"     : "MyDatabase",
  "engine"     : "mysql"
}
```

For database secrets, if you want to turn on automatic rotation, the secret must contain connection information for the database in the correct JSON structure. For more information, see the section called "JSON structure of a secret".

Metadata

A secret's metadata includes:

- An Amazon Resource Name (ARN) with the following format:

```
arn:aws:secretsmanager:<Region>:<AccountId>:secret:SecretName-6RandomCharacters
```

  Secrets Manager includes six random characters at the end of the secret name to help ensure that the secret ARN is unique. If the original secret is deleted, and then a new secret is created with the same name, the two secrets have different ARNs because of these characters. Users with access to the old secret don't automatically get access to the new secret because the ARNs are different.

- The name of the secret, a description, a resource policy, and tags.
- The ARN for an *encryption key*, an AWS KMS key that Secrets Manager uses to encrypt and decrypt the secret value. Secrets Manager stores secret text in an encrypted form and encrypts the secret in transit. See the section called “Secret encryption and decryption”.
- Information about how to rotate the secret, if you set up rotation. See *Rotate secrets*.

Secrets Manager uses IAM permissions policies to make sure that only authorized users can access or modify a secret. See Authentication and access control for AWS Secrets Manager.

A secret has *versions* that hold copies of the encrypted secret value. When you change the secret value, or the secret is rotated, Secrets Manager creates a new version. See the section called “Secret versions”.

You can use a secret across multiple AWS Regions by *replicating* it. When you replicate a secret, you create a copy of the original or *primary secret* called a *replica secret*. The replica secret remains linked to the primary secret. See *Multi-region replication*.

See *Manage secrets*.

Secret versions

A secret has *versions* that hold copies of the encrypted secret value. When you change the secret value, or the secret is rotated, Secrets Manager creates a new version.

Secrets Manager doesn't store a linear history of secrets with versions. Instead, it keeps track of three specific versions by labeling them:

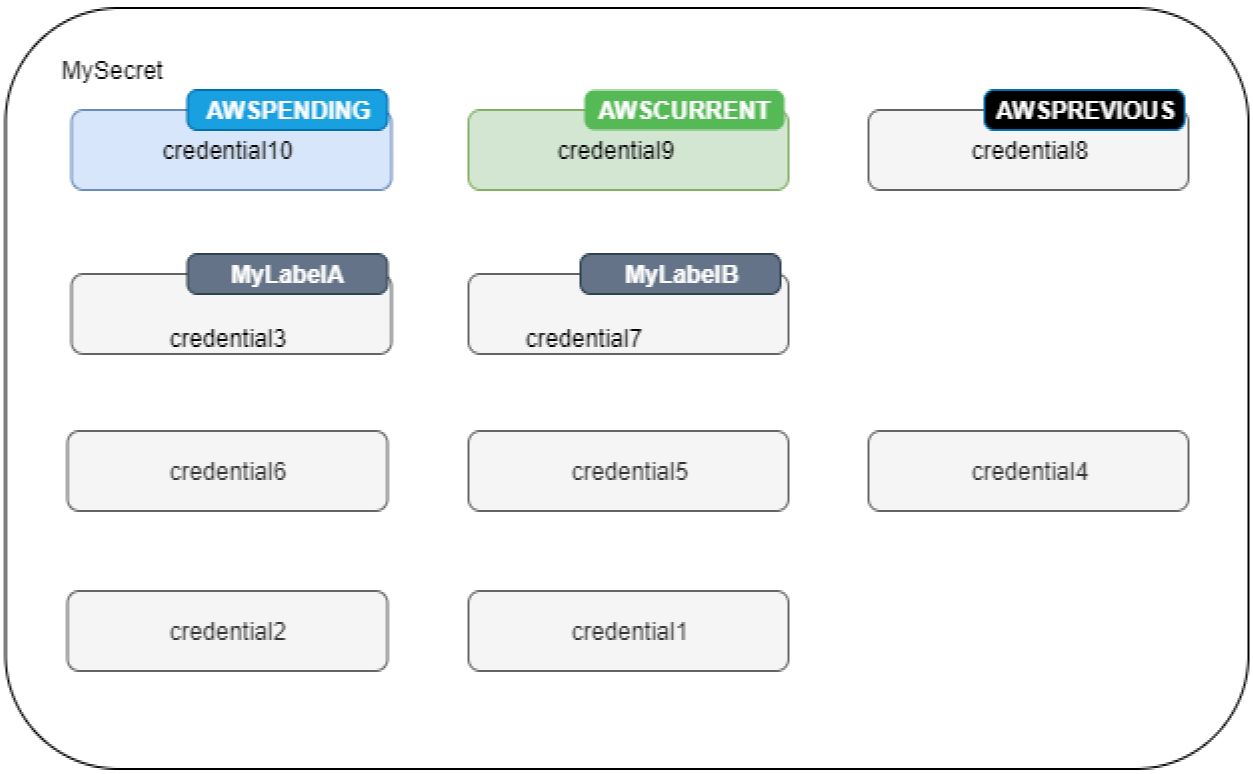
- The current version – `AWSCURRENT`
- The previous version – `AWSPREVIOUS`
- The pending version (during rotation) – `AWSPENDING`

A secret always has a version labeled AWSCURRENT, and Secrets Manager returns that version by default when you retrieve the secret value.

You can also label versions with your own labels by calling [update-secret-version-stage]() in the AWS CLI. You can attach up to 20 labels to versions in a secret. Two versions of a secret can't have the same staging label. Versions can have multiple labels.

Secrets Manager never removes labeled versions, but unlabeled versions are considered deprecated. Secrets Manager removes deprecated versions when there are more than 100. Secrets Manager doesn't remove versions created less than 24 hours ago.

The following figure shows a secret that has AWS labeled versions and customer labeled versions. The versions without labels are considered deprecated and will be removed by Secrets Manager at some point in the future.



JSON structure of AWS Secrets Manager secrets

You can store any text or binary in a Secrets Manager secret up to the maximum size of 65,536 Bytes.

If you use [the section called "Rotation by Lambda function"](), a secret must contain specific JSON fields that the rotation function expects. For example, for a secret that contains database credentials, the rotation function connects to the database to update credentials, so the secret must contain the database connection information.

If you use the console to edit rotation for a database secret, the secret must contain specific JSON key-value pairs that identify the database. Secrets Manager uses these fields to query the database to find the correct VPC to store a rotation function in.

JSON key names are case-sensitive.

Topics

Amazon RDS and Aurora credentials

To use the rotation function templates provided by Secrets Manager, use the following JSON structure. You can add more key/value pairs, for example to contain connection information for replica databases in other Regions.

DB2

For Amazon RDS Db2 instances, because users can't change their own passwords, you must provide admin credentials in a separate secret.

```
{
  "engine": "db2",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to None>",
  "port": <TCP port number. If not specified, defaults to 3306>,
  "masterarn": "<ARN of the elevated secret>",
  "dbInstanceIdentifier": <optional: ID of the instance. Alternately, use
 dbClusterIdentifier. Required for configuring rotation in the console.>",
```

```
  "dbClusterIdentifier": <optional: ID of the cluster. Alternately, use
 dbInstanceIdentifier. Required for configuring rotation in the console.>"
}
```

MariaDB

```
{
  "engine": "mariadb",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to None>",
  "port": <TCP port number. If not specified, defaults to 3306>,
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called “Alternating users”.>",
  "dbInstanceIdentifier": <optional: ID of the instance. Alternately, use
 dbClusterIdentifier. Required for configuring rotation in the console.>",
  "dbClusterIdentifier": <optional: ID of the cluster. Alternately, use
 dbInstanceIdentifier. Required for configuring rotation in the console.>"
}
```

MySQL

```
{
  "engine": "mysql",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to None>",
  "port": <TCP port number. If not specified, defaults to 3306>,
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called “Alternating users”.>",
  "dbInstanceIdentifier": <optional: ID of the instance. Alternately, use
 dbClusterIdentifier. Required for configuring rotation in the console.>",
  "dbClusterIdentifier": <optional: ID of the cluster. Alternately, use
 dbInstanceIdentifier. Required for configuring rotation in the console.>"
}
```

Oracle

```
{
  "engine": "oracle",
```

```
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name>",
  "port": <TCP port number. If not specified, defaults to 1521>,
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called “Alternating users”.>",
  "dbInstanceIdentifier": <optional: ID of the instance. Alternately, use
 dbClusterIdentifier. Required for configuring rotation in the console.>",
  "dbClusterIdentifier": <optional: ID of the cluster. Alternately, use
 dbInstanceIdentifier. Required for configuring rotation in the console.>"
}
```

Postgres

```
{
  "engine": "postgres",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to 'postgres'>",
  "port": <TCP port number. If not specified, defaults to 5432>,
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called “Alternating users”.>",
  "dbInstanceIdentifier": <optional: ID of the instance. Alternately, use
 dbClusterIdentifier. Required for configuring rotation in the console.>",
  "dbClusterIdentifier": <optional: ID of the cluster. Alternately, use
 dbInstanceIdentifier. Required for configuring rotation in the console.>"
}
```

SQLServer

```
{
  "engine": "sqlserver",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to 'master'>",
  "port": <TCP port number. If not specified, defaults to 1433>,
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called “Alternating users”.>",
  "dbInstanceIdentifier": <optional: ID of the instance. Alternately, use
 dbClusterIdentifier. Required for configuring rotation in the console.>",
```

```
  "dbClusterIdentifier": <optional: ID of the cluster.Alternately, use
 dbInstanceIdentifier.  Required for configuring rotation in the console.>"
}
```

Amazon Redshift credentials

To use the rotation function templates provided by Secrets Manager, use the following JSON structure. You can add more key/value pairs, for example to contain connection information for replica databases in other Regions.

```
{
  "engine": "redshift",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to None>",
  "dbClusterIdentifier": "<optional: database ID. Required for configuring rotation in
 the console.>"
  "port": <optional: TCP port number. If not specified, defaults to 5439>
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called "Alternating users".>"
}
```

Amazon Redshift Serverless credentials

To use the rotation function templates provided by Secrets Manager, use the following JSON structure. You can add more key/value pairs, for example to contain connection information for replica databases in other Regions.

```
{
  "engine": "redshift",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to None>",
  "namespaceName": "<optional: namespace name, Required for configuring rotation in the
 console.> "
  "port": <optional: TCP port number. If not specified, defaults to 5439>
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called "Alternating users".>"
```

```
}
```

Amazon DocumentDB credentials

To use the rotation function templates provided by Secrets Manager, use the following JSON structure. You can add more key/value pairs, for example to contain connection information for replica databases in other Regions.

```
{
  "engine": "mongo",
  "host": "<instance host name/resolvable DNS name>",
  "username": "<username>",
  "password": "<password>",
  "dbname": "<database name. If not specified, defaults to None>",
  "port": <TCP port number. If not specified, defaults to 27017>,
  "ssl": <true|false. If not specified, defaults to false>,
  "masterarn": "<optional: ARN of the elevated secret. Required for the the section
 called “Alternating users”.>",
  "dbClusterIdentifier": "<optional: database cluster ID. Alternately, use
 dbInstanceIdentifier. Required for configuring rotation in the console.>"
  "dbInstanceIdentifier": "<optional: database instance ID. Alternately, use
 dbClusterIdentifier. Required for configuring rotation in the console.>"
}
```

Amazon Timestream for InfluxDB secret structure

To rotate Timestream secrets, you can use the the section called “Amazon Timestream for InfluxDB” rotation templates.

For more information, see How Amazon Timestream for InfluxDB uses secrets in the *Amazon Timestream Developer Guide*.

The Timestream secrets must be in the correct JSON structure to be able to use the rotation templates. For more information, see What's in the secret in the *Amazon Timestream Developer Guide*.

Amazon ElastiCache credentials

The following example shows the JSON structure for a secret that stores ElastiCache credentials.

```
{
```

```
  "password": "<password>",
  "username": "<username>"
  "user_arn": "ARN of the Amazon EC2 user"
}
```

For more information, see Automatically rotating passwords for users in the *Amazon ElastiCache User Guide*.

Active Directory credentials

AWS Directory Service uses secrets to store Active Directory credentials. For more information, see Seamlessly join an Amazon EC2 Linux instance to your Managed AD Active Directory in the *AWS Directory Service Administration Guide*. Seamless domain join requires the key names in the following examples. If you don't use seamless domain join, you can change the names of the keys in the secret using environment variables as described in the rotation function template code.

To rotate Active Directory secrets, you can use the Active Directory rotation templates.

Active Directory credential

```
{
  "awsSeamlessDomainUsername": "<username>",
  "awsSeamlessDomainPassword": "<password>"
}
```

If you want to rotate the secret, you include the domain directory ID.

```
{
  "awsSeamlessDomainDirectoryId": "d-12345abc6e",
  "awsSeamlessDomainUsername": "<username>",
  "awsSeamlessDomainPassword": "<password>"
}
```

If the secret is used in conjunction with a secret that contains a keytab, you include the keytab secret ARNs.

```
{
  "awsSeamlessDomainDirectoryId": "d-12345abc6e",
  "awsSeamlessDomainUsername": "<username>",
  "awsSeamlessDomainPassword": "<password>",
```

```
  "directoryServiceSecretVersion": 1,
  "schemaVersion": "1.0",
  "keytabArns": [
    "<ARN of child keytab secret 1>,
    "<ARN of child keytab secret 2>,
    "<ARN of child keytab secret 3>,
  ],
  "lastModifiedDateTime": "2021-07-19 17:06:58"
}
```

Active Directory keytab

For information about using keytab files to authenticate to Active Directory accounts on Amazon EC2, see [Deploying and configuring Active Directory authentication with SQL Server 2017 on Amazon Linux 2](#).

```
{
  "awsSeamlessDomainDirectoryId": "d-12345abc6e",
  "schemaVersion": "1.0",
  "name": "< name>",
  "principals": [
    "aduser@MY.EXAMPLE.COM",
    "MSSQLSvc/test:1433@MY.EXAMPLE.COM"
  ],
  "keytabContents": "<keytab>",
  "parentSecretArn": "<ARN of parent secret>",
  "lastModifiedDateTime": "2021-07-19 17:06:58"
  "version": 1
}
```

Manage secrets with AWS Secrets Manager

Topics

Update the value for an AWS Secrets Manager secret

To update the value of your secret, you can use the console, the CLI, or an SDK. When you update the secret value, Secrets Manager creates a new version of the secret with the staging label `AWSCURRENT`. You can still access the old version, which has the label `AWSPREVIOUS`. You can also add your own labels. For more information, see Secrets Manager versioning.

To update the secret value (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. From the list of secrets, choose your secret.
3. On the secret details page, on the **Overview** tab, in the **Secret value** section, choose **Retrieve secret value** and then choose **Edit**.

AWS CLI

To update the secret value (AWS CLI)

- When you enter commands in a command shell, there is a risk of the command history being accessed or utilities having access to your command parameters. See the section called "Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets".

  The following `put-secret-value` creates a new version of a secret with two key-value pairs.

  ```
  aws secretsmanager put-secret-value \
        --secret-id MyTestSecret \
        --secret-string "{\"user\":\"diegor\",\"password\":\"EXAMPLE-PASSWORD\"}"
  ```

  The following `put-secret-value` creates a new version with a custom staging label. The new version will have the labels `MyLabel` and `AWSCURRENT`.

  ```
  aws secretsmanager put-secret-value \
        --secret-id MyTestSecret \
        --secret-string "{\"user\":\"diegor\",\"password\":\"EXAMPLE-PASSWORD\"}"
        --version-stages "MyLabel"
  ```

AWS SDK

We recommend you avoid calling `PutSecretValue` or `UpdateSecret` at a sustained rate of more than once every 10 minutes. When you call `PutSecretValue` or `UpdateSecret` to update the secret value, Secrets Manager creates a new version of the secret. Secrets Manager removes unlabeled versions when there are more than 100, but it does not remove versions created less than 24 hours ago. If you update the secret value more than once every 10 minutes, you create more versions than Secrets Manager removes, and you will reach the quota for secret versions.

To update a secret value, use the following actions: `UpdateSecret` or `PutSecretValue`. For more information, see the section called "AWS SDKs".

Generate a password with Secrets Manager

A common pattern for using Secrets Manager is to generate a password in Secrets Manager and then use that password in your database or service. You can do this using the following methods:

- CloudFormation – See *CloudFormation*.
- AWS CLI – See `get-random-password`.
- AWS SDKs – See `GetRandomPassword`.

Roll back a secret to a previous version

You can revert a secret to a previous version by moving the labels attached to secret versions using the AWS CLI. For information about how Secrets Manager stores versions of secrets, see the section called “Secret versions”.

The following `update-secret-version-stage` example moves the AWSCURRENT staging label to the previous version of a secret, which reverts the secret to the previous version. To find the ID for the previous version, use `list-secret-version-ids` or view the versions in the Secrets Manager console.

For this example, the version with the AWSCURRENT label is a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 and the version with the AWSPREVIOUS label is a1b2c3d4-5678-90ab-cdef-EXAMPLE22222. In this example, you move the AWSCURRENT label from version 11111 to 22222. Because the AWSCURRENT label is removed from a version, `update-secret-version-stage` automatically moves the AWSPREVIOUS label to that version (11111). The effect is that the AWSCURRENT and AWSPREVIOUS versions are swapped.

```
aws secretsmanager update-secret-version-stage \
  --secret-id MyTestSecret \
  --version-stage AWSCURRENT \
  --move-to-version-id a1b2c3d4-5678-90ab-cdef-EXAMPLE22222 \
  --remove-from-version-id a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
```

Change the encryption key for an AWS Secrets Manager secret

Secrets Manager uses envelope encryption with AWS KMS keys and data keys to protect each secret value. For each secret, you can choose which KMS key to use. You can use the AWS managed key **aws/secretsmanager**, or you can use a customer managed key. For most cases, we recommend using **aws/secretsmanager**, and there is no cost for using it. If you need to access the secret from another AWS account, or if you want to use your own KMS key so that you can rotate it or apply a key policy to it, use a customer managed key. You must have the section called “Permissions for the KMS key”. For information about the costs of using a customer managed key, see Pricing.

You can change the encryption key for your secret. For example, if you want to access the secret from another account, and the secret is currently encrypted using the AWS managed key `aws/secretsmanager`, you can switch to a customer managed key.

> **Tip**
>
> If you want to rotate your customer managed key, we recommend using AWS KMS automatic key rotation. For more information, see Rotating AWS KMS keys.

When you change the encryption key, Secrets Manager re-encrypts `AWSCURRENT`, `AWSPENDING`, and `AWSPREVIOUS` versions with the new key. To avoid locking you out of the secret, Secrets Manager keeps all existing versions encrypted with the previous key. That means you can decrypt `AWSCURRENT`, `AWSPENDING`, and `AWSPREVIOUS` versions with the previous key or the new key. If you don't have `kms:Decrypt` permission to the previous key, when you change the encryption key, Secrets Manager can't decrypt the secret versions to re-encrypt them. In this case, the existing versions are not re-encrypted.

To make it so `AWSCURRENT` can only be decrypted by the new encryption key, create a new version of the secret with the new key. Then to be able to decrypt the `AWSCURRENT` secret version, you must have permission to the new key.

If you deactivate the previous encryption key, you will not be able to decrypt any secret versions except `AWSCURRENT`, `AWSPENDING`, and `AWSPREVIOUS`. If you have other labelled secret versions that you want to retain access to, you need to recreate those versions with the new encryption key using the the section called "AWS CLI".

To change the encryption key for a secret (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. From the list of secrets, choose your secret.
3. On the secret details page, in the **Secrets details** section, choose **Actions**, and then choose **Edit encryption key**.

AWS CLI

If you change the encryption key for a secret and then deactivate the previous encryption key, you will not be able to decrypt any secret versions except `AWSCURRENT`, `AWSPENDING`, and

AWSPREVIOUS. If you have other labelled secret versions that you want to retain access to, you need to recreate those versions with the new encryption key using the the section called "AWS CLI".

To change the encryption key for a secret (AWS CLI)

1. The following update-secret example updates the KMS key used to encrypt the secret value. The KMS key must be in the same region as the secret.

```
aws secretsmanager update-secret \
      --secret-id MyTestSecret \
      --kms-key-id arn:aws:kms:us-west-2:123456789012:key/EXAMPLE1-90ab-cdef-fedc-
ba987EXAMPLE
```

2. (Optional) If you have secret versions that have custom labels, to re-encrypt them using the new key, you must recreate those versions.

   When you enter commands in a command shell, there is a risk of the command history being accessed or utilities having access to your command parameters. See the section called "Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets".

   a. Get the value of the secret version.

   ```
   aws secretsmanager get-secret-value \
         --secret-id MyTestSecret \
         --version-stage MyCustomLabel
   ```

   Make a note of the secret value.

   b. Create a new version with that value.

   ```
   aws secretsmanager put-secret-value \
       --secret-id testDescriptionUpdate \
       --secret-string "SecretValue" \
       --version-stages "MyCustomLabel"
   ```

Modify an AWS Secrets Manager secret

You can modify the metadata of a secret after it is created, depending on who created the secret. For secrets created by other services, you might need to use the other service to update or rotate it.

To determine who manages a secret, you can review the secret name. Secrets managed by other services are prefixed with the ID of that service. Or, in the AWS CLI, call describe-secret, and then review the field `OwningService`. For more information, see *Secrets managed by other services*.

For secrets you manage, you can modify the description, resource-based policy, the encryption key, and tags. You can also change the encrypted secret value; however, we recommend you use rotation to update secret values that contain credentials. Rotation updates both the secret in Secrets Manager and the credentials on the database or service. This keeps the secret automatically synchronized so when clients request a secret value, they always get a working set of credentials. For more information, see *Rotate secrets*.

Secrets Manager generates a CloudTrail log entry when you modify a secret. For more information, see the section called "Log with AWS CloudTrail ".

To update a secret you manage (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. From the list of secrets, choose your secret.
3. On the secret details page, do any of the following:

   Note that you can't change the name or ARN of a secret.

   - To update the description, in the **Secrets details** section, choose **Actions**, and then choose **Edit description**.
   - To update the encryption key, see the section called "Change the encryption key for a secret".
   - To update tags, on the **Tags** tab, choose **Edit tags**. See the section called "Tag secrets".
   - To update the secret value, see the section called "Update a secret value".
   - To update permissions for your secret, on the **Overview** tab, choose **Edit permissions**. See the section called "Resource-based policies".

   - To update rotation for your secret, on the **Rotation** tab, choose **Edit rotation**. See *Rotate secrets*.
   - To replicate your secret to other Regions, see Multi-region replication.
   - If your secret has replicas, you can change the encryption key for a replica. On the **Replication** tab, select the radio button for the replica, and then on the **Actions** menu, choose **Edit encryption key**. See the section called "Secret encryption and decryption".

- To change a secret so that it is managed by another service, you need to recreate the secret in that service. See Secrets managed by other services.

AWS CLI

Example Update secret description

The following `update-secret` example updates the description of a secret.

```
aws secretsmanager update-secret \
    --secret-id MyTestSecret \
    --description "This is a new description for the secret."
```

AWS SDK

We recommend you avoid calling `PutSecretValue` or `UpdateSecret` at a sustained rate of more than once every 10 minutes. When you call `PutSecretValue` or `UpdateSecret` to update the secret value, Secrets Manager creates a new version of the secret. Secrets Manager removes unlabeled versions when there are more than 100, but it does not remove versions created less than 24 hours ago. If you update the secret value more than once every 10 minutes, you create more versions than Secrets Manager removes, and you will reach the quota for secret versions.

To update a secret, use the following actions: `UpdateSecret` or `ReplicateSecretToRegions`. For more information, see the section called "AWS SDKs".

Find secrets in AWS Secrets Manager

When you search for secrets without a filter, Secrets Manager matches keywords in the secret name, description, tag key, and tag value. Searching without filters is not case-sensitive and ignores special characters, such as space, /, _, =, #, and only uses numbers and letters. When you search without a filter, Secrets Manager analyzes the search string to convert it to separate words. The words are separated by any change from uppercase to lowercase, from letter to number, or from number/letter to punctuation. For example, entering the search term `credsDatabase#892` searches for `creds`, `Database`, and `892` in name, description, and tag key and value.

Secrets Manager generates a CloudTrail log entry when you list secrets. For more information, see the section called "Log with AWS CloudTrail ".

Secrets Manager is a regional service and only secrets within the selected region are returned.

Search filters

If you don't use any filters, Secrets Manager breaks the search string into words and then searches all attributes for matches. This search is not case-sensitive. For example, searching for **`My_Secret`** matches secrets with the word **my** or **secret** in the name, description, or tags.

You can apply the following filters to your search:

Name

Matches the beginning of secret names; case-sensitive. For example, **Name: `Data`** returns a secret named `DatabaseSecret`, but not `databaseSecret` or `MyData`.

Description

Matches the words in secret descriptions, not case-sensitive. For example, **Description**: **`My Description`** matches secrets with the following descriptions:

- `My Description`
- `my description`
- `My basic description`
- `Description of my secret`

Managed by

Finds secrets managed by services outside of AWS, for example:

- 1Password
- Akeyless
- CyberArk
- HashiCorp

Owning service

Matches the beginning of the managing service ID prefix, not case-sensitive. For example, **`my-ser`** matches secrets managed by services with the prefix `my-serv` and `my-service`. For more information, see Secrets managed by other services.

Replicated secrets

You can filter for primary secrets, replica secrets, or secrets that aren't replicated.

Tag keys

Matches the beginning of tag keys; case-sensitive. For example, **Tag key: `Prod`** returns secrets with the tag `Production` and `Prod1`, but not secrets with the tag `prod` or `1 Prod`.

Tag values

Matches the beginning of tag values; case-sensitive. For example, **Tag value: `Prod`** returns secrets with the tag `Production` and `Prod1`, but not secrets with the tag value `prod` or `1 Prod`.

AWS CLI

Example List the secrets in your account

The following [`list-secrets`] example gets a list of the secrets in your account.

```
aws secretsmanager list-secrets
```

Example Filter the list of secrets in your account

The following [`list-secrets`] example gets a list of the secrets in your account that have **Test** in the name. Filtering by name is case sensitive.

```
aws secretsmanager list-secrets \
    --filters Key="name",Values="Test"
```

Example Find secrets that are managed by other AWS services

The following [`list-secrets`] example gets a list of secrets managed by a service. You specify the service by ID. For more information, see Secrets managed by other services.

```
aws secretsmanager list-secrets \
    --filters Key="owning-service",Values="<service ID prefix>"
```

AWS SDK

To find secrets by using one of the AWS SDKs, use `ListSecrets`. For more information, see the section called “AWS SDKs”.

Delete an AWS Secrets Manager secret

Because of the critical nature of secrets, AWS Secrets Manager intentionally makes deleting a secret difficult. Secrets Manager does not immediately delete secrets. Instead, Secrets Manager immediately makes the secrets inaccessible and scheduled for deletion after a recovery window of a minimum of seven days. Until the recovery window ends, you can recover a secret you previously deleted. There is no charge for secrets that you have marked for deletion.

You can't delete a primary secret if it is replicated to other Regions. First delete the replicas, then delete the primary secret. When you delete a replica, it is deleted immediately.

You can't directly delete a version of a secret. Instead, you remove all staging labels from the version using the AWS CLI or AWS SDK. This marks the version as deprecated, and then Secrets Manager can automatically delete the version in the background.

If you don't know whether an application still uses a secret, you can create an Amazon CloudWatch alarm to alert you to any attempts to access a secret during the recovery window. For more information, see Monitor when AWS Secrets Manager secrets scheduled for deletion are accessed.

To delete a secret, you must have `secretsmanager:ListSecrets` and `secretsmanager:DeleteSecret` permissions.

Secrets Manager generates a CloudTrail log entry when you delete a secret. For more information, see the section called “Log with AWS CloudTrail ”.

To delete a secret (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. In the list of secrets, choose the secret you want to delete.
3. In the **Secret details** section, choose **Actions**, and then choose **Delete secret**.
4. In the **Disable secret and schedule deletion** dialog box, in **Waiting period**, enter the number of days to wait before the deletion becomes permanent. Secrets Manager attaches a field called `DeletionDate` and sets the field to the current date and time, plus the number of days specified for the recovery window.
5. Choose **Schedule deletion**.

To view deleted secrets

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.

2. On the **Secrets** page, choose **Preferences** ( ).
3. In the Preferences dialog box, select **Show secrets scheduled for deletion**, and then choose **Save**.

To delete a replica secret

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose the primary secret.
3. In the **Replicate Secret** section, choose the replica secret.
4. From the **Actions** menu, choose **Delete Replica**.

AWS CLI

Example Delete a secret

The following `delete-secret` example deletes a secret. You can recover the secret with `restore-secret` until the date and time in the DeletionDate response field. To delete a secret that is replicated to other regions, first remove its replicas with `remove-regions-from-replication`, and then call `delete-secret`.

```
aws secretsmanager delete-secret \
    --secret-id MyTestSecret \
    --recovery-window-in-days 7
```

Example Delete a secret immediately

The following `delete-secret` example deletes a secret immediately without a recovery window. You can't recover this secret.

```
aws secretsmanager delete-secret \
    --secret-id MyTestSecret \
    --force-delete-without-recovery
```

Example Delete a replica secret

The following [remove-regions-from-replication](#) example deletes a replica secret in eu-west-3. To delete a primary secret that is replicated to other regions, first delete the replicas and then call [delete-secret](#).

```
aws secretsmanager remove-regions-from-replication \
    --secret-id MyTestSecret \
    --remove-replica-regions eu-west-3
```

AWS SDK

To delete a secret, use the `DeleteSecret` command. To delete a version of a secret, use the `UpdateSecretVersionStage` command. To delete a replica, use the `StopReplicationToReplica` command. For more information, see the section called "AWS SDKs".

Restore an AWS Secrets Manager secret

Secrets Manager considers a secret scheduled for deletion *deprecated* and you can no longer directly access it. After the recovery window has passed, Secrets Manager deletes the secret permanently. Once Secrets Manager deletes the secret, you can't recover it. Before the end of the recovery window, you can recover the secret and make it accessible again. This removes the `DeletionDate` field, which cancels the scheduled permanent deletion.

To restore a secret and the metadata in the console, you must have `secretsmanager:ListSecrets` and `secretsmanager:RestoreSecret` permissions.

Secrets Manager generates a CloudTrail log entry when you restore a secret. For more information, see the section called "Log with AWS CloudTrail ".

To restore a secret (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. In the list of secrets, choose the secret you want to restore.

   If deleted secrets don't appear in your list of secrets, choose **Preferences** ( ).

In the Preferences dialog box, select **Show secrets scheduled for deletion**, and then choose **Save**.

3. On the **Secret details** page, choose **Cancel deletion**.
4. In the **Cancel secret deletion** dialog box, choose **Cancel deletion**.

AWS CLI

Example Restore a previously deleted secret

The following `restore-secret` example restores a secret that was previously scheduled for deletion.

```
aws secretsmanager restore-secret \
    --secret-id MyTestSecret
```

AWS SDK

To restore a secret marked for deletion, use the `RestoreSecret` command. For more information, see the section called “AWS SDKs”.

Tagging secrets in AWS Secrets Manager

In AWS Secrets Manager, you can assign metadata to your secrets using tags. A tag is a key-value pair that you define for a secret. Tags help you manage AWS resources and organize data, including billing information.

With tags, you can:

- Manage, search, and filter secrets and other resources in your AWS account
- Control access to secrets based on attached tags
- Track and categorize expenses associated with specific secrets or projects

For more information about using tags to control access, see the section called "Control access to secrets using tags".

To learn about cost allocation tags, see Using AWS cost allocation tags in the AWS Billing User Guide.

For information about tag quotas and naming restrictions, see Service quotas for Tagging in the *AWS General Reference guide*. Tags are case-sensitive.

Secrets Manager generates a CloudTrail log entry when you tag or untag a secret. For more information, see the section called "Log with AWS CloudTrail ".

Tip

Use a consistent tagging scheme across all your AWS resources. For best practices, see the Tagging Best Practices whitepaper.

Review tag basics

You can find secrets by tags in the console, AWS CLI, and SDKs. AWS also provides the Resource Groups tool to create a custom console that consolidates and organizes your resources based on their tags. To find secrets with a specific tag, see the section called "Find secrets".

You can use the Secrets Manager console, AWS CLI, or Secrets Manager API to:

- Create a secret with tags

- Add tags to a secret
- List the tags for your secrets
- Remove tags from a secret

You can use tags to categorize your secrets. For example, you can categorize secrets by purpose, owner, or environment. Because you define the key and value for each tag, you can create a custom set of categories to meet your specific needs. Here are several examples of tags:

- `Project: Project name`
- `Owner: Name`
- `Purpose: Load testing`
- `Application: Application name`
- `Environment: Production`

Track costs using tagging

You can use tags to categorize and track your AWS costs. When you apply tags to your AWS resources, including secrets, your AWS cost allocation report includes usage and costs aggregated by tags. You can apply tags that represent business categories (such as cost centers, application names, or owners) to organize your costs across multiple services. For more information, see Use Cost Allocation Tags for Custom Billing Reports in the *AWS Billing User Guide*.

Understand tag restrictions

The following restrictions apply to tags.

Basic restrictions

- The maximum number of tags per resource (secret) is 50.
- Tag keys and values are case-sensitive.
- You can't change or edit tags for a deleted secret.

Tag key restrictions

- Each tag key must be unique. If you add a tag with a key that's already in use, your new tag overwrites the existing key-value pair.

- You can't start a tag key with `aws:` because this prefix is reserved for use by AWS. AWS creates tags that begin with this prefix on your behalf, but you can't edit or delete them.
- Tag keys must be between 1 and 128 Unicode characters in length.
- Tag keys must consist of the following characters: Unicode letters, digits, white space, and the following special characters: `_ . / = + - @`.

Tag value restrictions

- Tag values must be between 0 and 255 Unicode characters in length.
- Tag values can be blank. Otherwise, they must consist of the following characters: Unicode letters, digits, white space, and any of the following special characters: `_ . / = + - @`.

Tag secrets using the Secrets Manager console

You can manage tags for your secrets using the [Secrets Manager console](https://console.aws.amazon.com/secretsmanager/).

To access the tagging features, do the following:

1. Open the Secrets Manager console.
2. In the navigation bar, choose your preferred Region.
3. On the **Secrets** page, select a secret.

To view the tags for a secret

- On the **Secret Details** page, choose the **Tags** tab.

To create a secret with a tag

- Follow the steps in [Create secrets](create_secret.md).

To add or edit tags for a secret

1. On the **Secret Details** page, choose the **Tags** tab and then choose **Edit tags**.
2. Enter the tag key in the **Key** field. Optionally, enter a tag value in the **Value** field.
3. Choose **Save**. The new or updated tag appears in the list of tags.

> **Note**
>
> If the **Save** button is not enabled, the tag key or value might not meet the tag restrictions. For more information, see Understand tag restrictions.

To remove a tag from a secret

1. On the **Secret details** page, choose the **Tags** tab, and then choose the **Remove** icon next to the tag you want to remove.
2. Choose **Save** to confirm the removal, or select **Undo** to cancel.

Tag secrets using the AWS CLI

AWS CLI examples

Example Add a tag to a secret

The following `tag-resource` example shows how to attach a tag with shorthand syntax.

```
aws secretsmanager tag-resource \
            --secret-id MyTestSecret \
            --tags Key=FirstTag,Value=FirstValue
```

Example Add multiple tags to a secret

The following `tag-resource` example attaches two key-value tags to a secret.

```
aws secretsmanager tag-resource \
            --secret-id MyTestSecret \
            --tags '[{"Key": "FirstTag", "Value": "FirstValue"}, {"Key": "SecondTag",
 "Value": "SecondValue"}]'
```

Example Remove tags from a secret

The following `untag-resource` example removes two tags from a secret. For each tag, both key and value are removed.

```
aws secretsmanager untag-resource \
```

```
            --secret-id MyTestSecret \
            --tag-keys '[ "FirstTag", "SecondTag"]'
```

Tag secrets using the Secrets Manager API

You can add, list, and remove tags using the Secrets Manager API. For examples, see the following documentation:

- ListSecrets: Use `ListSecrets` to view the tags applied to a secret
- TagResource: Add tags to a secret
- Untag: Remove tags from a secret

Tag secrets using the Secrets Manager AWS SDK

To change tags for your secret, use the following API operations:

- ListSecrets: Use `ListSecrets` to view the tags applied to a secret
- `TagResource`: Add tags to a secret
- `UntagResource`: Remove tags from a secret

For more information about using the SDK, see the section called “AWS SDKs”.

Replicate AWS Secrets Manager secrets across Regions

You can replicate your secrets in multiple AWS Regions to support applications spread across those Regions to meet Regional access and low latency requirements. If you later need to, you can promote a replica secret to a standalone and then set it up for replication independently. Secrets Manager replicates the encrypted secret data and metadata such as tags and resource policies across the specified Regions.

The ARN for a replicated secret is the same as the primary secret except for the Region, for example:

- Primary secret: `arn:aws:secretsmanager:Region1:123456789012:secret:MySecret-a1b2c3`
- Replica secret: `arn:aws:secretsmanager:Region2:123456789012:secret:MySecret-a1b2c3`

For pricing information for replica secrets, see AWS Secrets Manager Pricing.

When you store database credentials for a source database that is replicated to other Regions, the secret contains connection information for the source database. If you then replicate the secret, the replicas are copies of the source secret and contain the same connection information. You can add additional key/value pairs to the secret for regional connection information.

If you turn on rotation for your primary secret, Secrets Manager rotates the secret in the primary Region, and the new secret value propagates to all of the associated replica secrets. You don't have to manage rotation individually for all of the replica secrets.

You can replicate secrets across all of your enabled AWS Regions. However, if you use Secrets Manager in special AWS Regions such as AWS GovCloud (US) or China Regions, you can only configure secrets and the replicas within these specialized AWS Regions. You can't replicate a secret in your enabled AWS Regions to a specialized Region or replicate secrets from a specialized region to a commercial region.

Before you can replicate a secret to another Region, you must enable that Region. For more information, see Managing AWS Regions.

It is possible to use a secret across multiple Regions without replicating it by calling the Secrets Manager endpoint in the Region where the secret is stored. For a list of endpoints, see the section called "Secrets Manager endpoints". To use replication to improve your workload's resilience, see Disaster Recovery (DR) Architecture on AWS, Part I: Strategies for Recovery in the Cloud.

Secrets Manager generates a CloudTrail log entry when you replicate a secret. For more information, see the section called "Log with AWS CloudTrail ".

To replicate a secret to other Regions (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. From the list of secrets, choose your secret.
3. On the secret details page, on the **Replication** tab, do one of the following:
   - If your secret is not replicated, choose **Replicate secret**.
   - If your secret is replicated, in the **Replicate secret** section, choose **Add Region**.
4. In the **Add replica regions** dialog box, do the following:
   a. For **AWS Region**, choose the Region you want to replicate the secret to.
   b. (Optional) For **Encryption key**, choose a KMS key to encrypt the secret with. The key must be in the replica Region.
   c. (Optional) To add another Region, choose **Add more regions**.
   d. Choose **Replicate**.

   You return to the secret details page. In the **Replicate secret** section, the **Replication status** shows for each Region.

AWS CLI

Example Replicate a secret to another region

The following `replicate-secret-to-regions` example replicates a secret to eu-west-3. The replica is encrypted with the AWS managed key **aws/secretsmanager**.

```
aws secretsmanager replicate-secret-to-regions \
        --secret-id MyTestSecret \
        --add-replica-regions Region=eu-west-3
```

Example Create a secret and replicate it

The following example creates a secret and replicates it to eu-west-3. The replica is encrypted with the AWS managed key **aws/secretsmanager**.

```
aws secretsmanager create-secret \
    --name MyTestSecret \
    --description "My test secret created with the CLI." \
    --secret-string "{\"user\":\"diegor\",\"password\":\"EXAMPLE-PASSWORD\"}"
    --add-replica-regions Region=eu-west-3
```

AWS SDK

To replicate a secret, use the `ReplicateSecretToRegions` command. For more information, see the section called “AWS SDKs”.

Promote a replica secret to a standalone secret in AWS Secrets Manager

A replica secret is a secret that is replicated from a primary in another AWS Region. It has the same secret value and metadata as the primary, but it can be encrypted with a different KMS key. A replica secret can't be updated independently from its primary secret, except for its encryption key. Promoting a replica secret disconnects the replica secret from the primary secret and makes the replica secret a standalone secret. Changes to the primary secret won't replicate to the standalone secret.

You might want to promote a replica secret to a standalone secret as a disaster recovery solution if the primary secret becomes unavailable. Or you might want to promote a replica to a standalone secret if you want to turn on rotation for the replica.

If you promote a replica, be sure to update the corresponding applications to use the standalone secret.

Secrets Manager generates a CloudTrail log entry when you promote a secret. For more information, see the section called “Log with AWS CloudTrail ”.

To promote a replica secret (console)

1. Log in to the Secrets Manager at https://console.aws.amazon.com/secretsmanager/.

2. Navigate to the replica region.
3. On the **Secrets** page, choose the replica secret.
4. On the replica secret details page, choose **Promote to standalone secret**.
5. In the **Promote replica to standalone secret** dialog box, enter the Region and then choose **Promote replica**.

AWS CLI

Example Promote a replica secret to a primary

The following `stop-replication-to-replica` example removes the link between a replica secret to the primary. The replica secret is promoted to a primary secret in the replica region. You must call `stop-replication-to-replica` from within the replica region.

```
aws secretsmanager stop-replication-to-replica \
    --secret-id MyTestSecret
```

AWS SDK

To promote a replica to a standalone secret, use the `StopReplicationToReplica` command. You must call this command from the replica secret Region. For more information, see the section called “AWS SDKs”.

Prevent AWS Secrets Manager replication

Because secrets can be replicated using `ReplicateSecretToRegions` or when they are created using `CreateSecret`, if you want to prevent users from replicating secrets, we recommend you prevent actions that contain the `AddReplicaRegions` parameter. You can use a `Condition` statement in your permission policies to only allow actions that don't add replica regions. See the following policy examples for Condition statements you can use.

Example Prevent replication permission

The following policy example shows how to allow all actions that don't add replica regions. This prevents users from replicating secrets through both `ReplicateSecretToRegions` and `CreateSecret`.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "secretsmanager:*",
      "Resource": "*",
      "Condition": {
        "Null": {
          "secretsmanager:AddReplicaRegions": "true"
        }
      }
    }
  ]
}
```

Example Allow replication permission only to specific Regions

The following policy shows how to allow all of the following:

- Create secrets without replication
- Create secrets with replication to Regions only in United States and Canada
- Replicate secrets to Regions only in United States and Canada

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:CreateSecret",
        "secretsmanager:ReplicateSecretToRegions"
      ],
      "Resource": "*",
      "Condition": {
```

```
          "ForAllValues:StringLike": {
            "secretsmanager:AddReplicaRegions": [
              "us-*",
              "ca-*"
            ]
          }
        }
      }
    ]
}
```

Troubleshoot AWS Secrets Manager replication

AWS Secrets Manager replication might fail for various reasons. To check why a secret failed to replicate, you can do one of the following:

- Call the `DescribeSecret` API operation
- Review AWS CloudTrail events

When replication fails:

- If there are no usable secret versions, Secrets Manager removes the secret from the replica Region.
- If there are successfully replicated secret versions, they remain in the replica Region until you explicitly remove them using the `RemoveRegionsFromReplication` API operation.

The following sections describe some common reasons for replication failures.

A secret with the same name exists in the selected Region

To resolve this issue, you can overwrite the duplicate name secret in the replica Region. Retry replication, and then in the **Retry replication** dialog box, choose **Overwrite**.

No permissions available on the KMS key to complete the replication

Secrets Manager first decrypts the secret before re-encrypting with the new KMS key in the replica Region. If you don't have `kms:Decrypt` permission to the encryption key in the primary Region, you will encounter this error. To encrypt the replicated secret with a KMS key other than `aws/`

`secretsmanager`, you need `kms:GenerateDataKey` and `kms:Encrypt` to the key. See the section called "Permissions for the KMS key".

The KMS key is disabled or not found

If the encryption key in the primary Region is disabled or deleted, Secrets Manager can't replicate the secret. This error can occur even if you have changed the encryption key, if the secret has custom labelled versions that were encrypted with the disabled or deleted encryption key. For information about how Secrets Manager does encryption, see the section called "Secret encryption and decryption". To work around this issue, you can recreate the secret versions so that Secrets Manager encrypts them with the current encryption key. For more information, see Change the encryption key for a secret. Then retry replication.

```
aws secretsmanager put-secret-value \
  --secret-id testDescriptionUpdate \
  --secret-string "SecretValue" \
  --version-stages "MyCustomLabel"
```

You have not enabled the Region where the replication occurs

For information about how to enable a Region, see Managing AWS Regions. in the *AWS Account Management Reference Guide*.

Get secrets from AWS Secrets Manager

Secrets Manager generates a CloudTrail log entry when you retrieve a secret. For more information, see the section called “Log with AWS CloudTrail ”.

You can retrieve secret values using:

- Get a Secrets Manager secret value using Java
- Get a Secrets Manager secret value using Python
- Get a Secrets Manager secret value using .NET
- Get a Secrets Manager secret value using Go
- Get a Secrets Manager secret value using Rust
- Use AWS Secrets Manager secrets in Amazon Elastic Kubernetes Service
- Use AWS Secrets Manager secrets in AWS Lambda functions
- Using the AWS Secrets Manager Agent
- Get a Secrets Manager secret value using the C++ AWS SDK
- Get a Secrets Manager secret value using the JavaScript AWS SDK
- Get a Secrets Manager secret value using the Kotlin AWS SDK
- Get a Secrets Manager secret value using the PHP AWS SDK
- Get a Secrets Manager secret value using the Ruby AWS SDK
- Get a secret value using the AWS CLI
- Get a secret value using the AWS console
- Use AWS Secrets Manager secrets in AWS Batch
- Get an AWS Secrets Manager secret in an CloudFormation resource
- Use AWS Secrets Manager secrets in GitHub jobs
- Use AWS Secrets Manager in GitLab
- Use AWS Secrets Manager secrets in AWS IoT Greengrass
- Use AWS Secrets Manager secrets in Parameter Store

Get a Secrets Manager secret value using Java

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

To connect to a database using the credentials in a secret, you can use the Secrets Manager SQL Connection drivers, which wrap the base JDBC driver. This also uses client-side caching, so it can reduce the cost for calling Secrets Manager APIs.

Topics

Get a Secrets Manager secret value using Java with client-side caching

When you retrieve a secret, you can use the Secrets Manager Java-based caching component to cache it for future use. Retrieving a cached secret is faster than retrieving it from Secrets Manager. Because there is a cost for calling Secrets Manager APIs, using a cache can reduce your costs. For all of the ways you can retrieve secrets, see Get secrets.

The cache policy is Least Recently Used (LRU), so when the cache must discard a secret, it discards the least recently used secret. By default, the cache refreshes secrets every hour. You can configure how often the secret is refreshed in the cache, and you can hook into the secret retrieval to add more functionality.

The cache does not force garbage collection once cache references are freed. The cache implementation does not include cache invalidation. The cache implementation is focused around the cache itself, and is not security hardened or focused. If you require additional security such as encrypting items in the cache, use the interfaces and abstract methods provided.

To use the component, you must have the following:

- A Java 8 or higher development environment. See Java SE Downloads on the Oracle website.

To download the source code, see Secrets Manager Java-based caching client component on GitHub.

To add the component to your project, in your Maven pom.xml file, include the following dependency. For more information about Maven, see the Getting Started Guide on the Apache Maven Project website.

```
<dependency>
  <groupId>com.amazonaws.secretsmanager</groupId>
  <artifactId>aws-secretsmanager-caching-java</artifactId>
  <version>1.0.2</version>
</dependency>
```

Required permissions:

- `secretsmanager:DescribeSecret`
- `secretsmanager:GetSecretValue`

For more information, see Permissions reference.

Reference

- SecretCache
- SecretCacheConfiguration
- SecretCacheHook

Example Retrieve a secret

The following code example shows a Lambda function that retrieves a secret string. It follows the best practice of instantiating the cache outside of the function handler, so it doesn't keep calling the API if you call the Lambda function again.

```
package com.amazonaws.secretsmanager.caching.examples;

    import com.amazonaws.services.lambda.runtime.Context;
    import com.amazonaws.services.lambda.runtime.RequestHandler;
    import com.amazonaws.services.lambda.runtime.LambdaLogger;

    import com.amazonaws.secretsmanager.caching.SecretCache;

    public class SampleClass implements RequestHandler<String, String> {
```

```
        private final SecretCache cache  = new SecretCache();

        @Override public String handleRequest(String secretId,  Context context) {
            final String secret  = cache.getSecretString(secretId);

          // Use the secret, return success;

        }
    }
```

SecretCache

An in-memory cache for secrets requested from Secrets Manager. You use the section called "getSecretString" or the section called "getSecretBinary" to retrieve a secret from the cache. You can configure the cache settings by passing in a the section called "SecretCacheConfiguration" object in the constructor.

For more information, including examples, see the section called "Java with client-side caching".

Constructors

`public SecretCache()`

Default constructor for a `SecretCache` object.

`public SecretCache(AWSSecretsManagerClientBuilder builder)`

Constructs a new cache using a Secrets Manager client created using the provided `AWSSecretsManagerClientBuilder`. Use this constructor to customize the Secrets Manager client, for example to use a specific Region or endpoint.

`public SecretCache(AWSSecretsManager client)`

Constructs a new secret cache using the provided `AWSSecretsManagerClient`. Use this constructor to customize the Secrets Manager client, for example to use a specific Region or endpoint.

`public SecretCache(SecretCacheConfiguration config)`

Constructs a new secret cache using the provided `the section called "SecretCacheConfiguration"`.

Methods

getSecretString

```
public String getSecretString(final String secretId)
```

Retrieves a string secret from Secrets Manager. Returns a `String`.

getSecretBinary

```
public ByteBuffer getSecretBinary(final String secretId)
```

Retrieves a binary secret from Secrets Manager. Returns a `ByteBuffer`.

refreshNow

```
public boolean refreshNow(final String secretId) throws
InterruptedException
```

Forces the cache to refresh. Returns `true` if the refresh completed without error, otherwise `false`.

close

```
public void close()
```

Closes the cache.

SecretCacheConfiguration

Cache configuration options for a the section called "SecretCache", such as max cache size and Time to Live (TTL) for cached secrets.

Constructor

```
public SecretCacheConfiguration
```

Default constructor for a `SecretCacheConfiguration` object.

Methods

getClient

```
public AWSSecretsManager getClient()
```

Returns the `AWSSecretsManagerClient` that the cache retrieves secrets from.

setClient

```
public void setClient(AWSSecretsManager client)
```

Sets the `AWSSecretsManagerClient` client that the cache retrieves secrets from.

getCacheHook

```
public SecretCacheHook getCacheHook()
```

Returns the the section called "SecretCacheHook" interface used to hook cache updates.

setCacheHook

```
public void setCacheHook(SecretCacheHook cacheHook)
```

Sets the the section called "SecretCacheHook" interface used to hook cache updates.

getMaxCacheSize

```
public int getMaxCacheSize()
```

Returns the maximum cache size. The default is 1024 secrets.

setMaxCacheSize

```
public void setMaxCacheSize(int maxCacheSize)
```

Sets the maximum cache size. The default is 1024 secrets.

getCacheItemTTL

```
public long getCacheItemTTL()
```

Returns the TTL in milliseconds for the cached items. When a cached secret exceeds this TTL, the cache retrieves a new copy of the secret from the `AWSSecretsManagerClient`. The default is 1 hour in milliseconds.

The cache refreshes the secret synchronously when the secret is requested after the TTL. If the synchronous refresh fails, the cache returns the stale secret.

setCacheItemTTL

```
public void setCacheItemTTL(long cacheItemTTL)
```

Sets the TTL in milliseconds for the cached items. When a cached secret exceeds this TTL, the cache retrieves a new copy of the secret from the `AWSSecretsManagerClient`. The default is 1 hour in milliseconds.

getVersionStage

```
public String getVersionStage()
```

Returns the version of secrets that you want to cache. For more information, see Secret versions. The default is `"AWSCURRENT"`.

setVersionStage

```
public void setVersionStage(String versionStage)
```

Sets the version of secrets that you want to cache. For more information, see Secret versions. The default is `"AWSCURRENT"`.

SecretCacheConfiguration withClient

```
public SecretCacheConfiguration withClient(AWSSecretsManager client)
```

Sets the `AWSSecretsManagerClient` to retrieve secrets from. Returns the updated `SecretCacheConfiguration` object with the new setting.

SecretCacheConfiguration withCacheHook

```
public SecretCacheConfiguration withCacheHook(SecretCacheHook cacheHook)
```

Sets the interface used to hook the in-memory cache. Returns the updated `SecretCacheConfiguration` object with the new setting.

SecretCacheConfiguration withMaxCacheSize

```
public SecretCacheConfiguration withMaxCacheSize(int maxCacheSize)
```

Sets the maximum cache size. Returns the updated `SecretCacheConfiguration` object with the new setting.

SecretCacheConfiguration withCacheItemTTL

```
public SecretCacheConfiguration withCacheItemTTL(long cacheItemTTL)
```

Sets the TTL in milliseconds for the cached items. When a cached secret exceeds this TTL, the cache retrieves a new copy of the secret from the `AWSSecretsManagerClient`. The default is 1 hour in milliseconds. Returns the updated `SecretCacheConfiguration` object with the new setting.

SecretCacheConfiguration withVersionStage

```
public SecretCacheConfiguration withVersionStage(String versionStage)
```

Sets the version of secrets that you want to cache. For more information, see Secret versions. Returns the updated `SecretCacheConfiguration` object with the new setting.

SecretCacheHook

An interface to hook into a the section called "SecretCache" to perform actions on the secrets being stored in the cache.

put

```
Object put(final Object o)
```

Prepare the object for storing in the cache.

Returns the object to store in the cache.

get

```
Object get(final Object cachedObject)
```

Derive the object from the cached object.

Returns the object to return from the cache

Connect to a SQL database using JDBC with credentials in an AWS Secrets Manager secret

In Java applications, you can use the Secrets Manager SQL Connection drivers to connect to MySQL, PostgreSQL, Oracle, MSSQLServer, Db2, and Redshift databases using credentials stored in Secrets Manager. Each driver wraps the base JDBC driver, so you can use JDBC calls to access your database. However, instead of passing a username and password for the connection, you provide the ID of a secret. The driver calls Secrets Manager to retrieve the secret value, and then uses the credentials in the secret to connect to the database. The driver also caches the credentials using the Java client-side caching library, so future connections don't require a call to Secrets Manager. By default, the cache refreshes every hour and also when the secret is rotated. To configure the cache, see the section called “SecretCacheConfiguration”.

You can download the source code from GitHub.

To use the Secrets Manager SQL Connection drivers:

- Your application must be in Java 8 or higher.
- Your secret must be one of the following:
  - A database secret in the expected JSON structure. To check the format, in the Secrets Manager console, view your secret and choose **Retrieve secret value**. Alternatively, in the AWS CLI, call get-secret-value.
  - An Amazon RDS managed secret. For this type of secret, you must specify an endpoint and port when you establish the connection.
  - An Amazon Redshift managed secret. For this type of secret, you must specify an endpoint and port when you establish the connection.

If your database is replicated to other Regions, to connect to a replica database in another Region, you specify the regional endpoint and port when you create the connection. You can store regional connection information in the secret as extra key/value pairs, in SSM Parameter Store parameters, or in your code configuration.

To add the driver to your project, in your Maven build file `pom.xml`, add the following dependency for the driver. For more information, see Secrets Manager SQL Connection Library on the Maven Central Repository website.

```
<dependency>
    <groupId>com.amazonaws.secretsmanager</groupId>
    <artifactId>aws-secretsmanager-jdbc</artifactId>
    <version>1.0.12</version>
</dependency>
```

The driver uses the *default credential provider chain*. If you run the driver on Amazon EKS, it might pick up the credentials of the node it is running on instead of the service account role. To address this, add version 1 of `com.amazonaws:aws-java-sdk-sts` to your Gradle or Maven project file as a dependency.

To set an AWS PrivateLink DNS endpoint URL and a region in the `secretsmanager.properties` file:

```
drivers.vpcEndpointUrl = endpoint URL
drivers.vpcEndpointRegion = endpoint region
```

To override the primary region, set the `AWS_SECRET_JDBC_REGION` environment variable or make the following change to the `secretsmanager.properties` file:

```
drivers.region = region
```

Required permissions:

- `secretsmanager:DescribeSecret`
- `secretsmanager:GetSecretValue`

For more information, see Permissions reference.

Examples:

- Establish a connection to a database
- Establish a connection by specifying the endpoint and port
- Use c3p0 connection pooling to establish a connection
- Use c3p0 connection pooling to establish a connection by specifying the endpoint and port

Establish a connection to a database

The following example shows how to establish a connection to a database using the credentials and connection information in a secret. Once you have the connection, you can use JDBC calls to access the database. For more information, see JDBC Basics on the Java documentation website.

MySQL

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerMySQLDriver" ).newInstance

// Retrieve the connection info from the secret using the secret ARN
String URL = "secretId";
```

```
// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

PostgreSQL

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerPostgreSQLDriver" ).newIns

// Retrieve the connection info from the secret using the secret ARN
String URL = "secretId";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

Oracle

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerOracleDriver" ).newInstanc

// Retrieve the connection info from the secret using the secret ARN
String URL = "secretId";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

MSSQLServer

```
// Load the JDBC driver
```

```
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerMSSQLServerDriver" ).newIn

// Retrieve the connection info from the secret using the secret ARN
String URL = "secretId";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

Db2

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerDb2Driver" ).newInstance()

// Retrieve the connection info from the secret using the secret ARN
String URL = "secretId";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

Redshift

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerRedshiftDriver" ).newInsta

// Retrieve the connection info from the secret using the secret ARN
String URL = "secretId";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
```

```
conn = DriverManager.getConnection(URL, info);
```

Establish a connection by specifying the endpoint and port

The following example shows how to establish a connection to a database using the credentials in a secret with an endpoint and port that you specify.

Amazon RDS managed secrets don't include the endpoint and port of the database. To connect to a database using master credentials in a secret that's managed by Amazon RDS, you specify them in your code.

Secrets that are replicated to other Regions can improve latency for the connection to the regional database, but they do not contain different connection information from the source secret. Each replica is a copy of the source secret. To store regional connection information in the secret, add more key/value pairs for the endpoint and port information for the Regions.

Once you have the connection, you can use JDBC calls to access the database. For more information, see JDBC Basics on the Java documentation website.

MySQL

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerMySQLDriver" ).newInstance

// Set the endpoint and port. You can also retrieve it from a key/value pair in the
 secret.
String URL = "jdbc-secretsmanager:mysql://example.com:3306";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

PostgreSQL

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerPostgreSQLDriver" ).newIns
```

```
// Set the endpoint and port. You can also retrieve it from a key/value pair in the
 secret.
String URL = "jdbc-secretsmanager:postgresql://example.com:5432/database";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

Oracle

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerOracleDriver" ).newInstanc

// Set the endpoint and port. You can also retrieve it from a key/value pair in the
 secret.
String URL = "jdbc-secretsmanager:oracle:thin:@example.com:1521/ORCL";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

MSSQLServer

```
// Load the JDBC driver
Class.forName( "com.amazonaws.secretsmanager.sql.AWSSecretsManagerMSSQLServerDriver" ).newIn

// Set the endpoint and port. You can also retrieve it from a key/value pair in the
 secret.
String URL = "jdbc-secretsmanager:sqlserver://example.com:1433";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );
```

```
// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

Db2

```
// Load the JDBC driver
Class.forName( "com.amazonaws.com.amazonaws.secretsmanager.sql.AWSSecretsManagerDb2Driver" )

// Set the endpoint and port. You can also retrieve it from a key/value pair in the
 secret.
String URL = "jdbc-secretsmanager:db2://example.com:50000";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

Redshift

```
// Load the JDBC driver
Class.forName( "com.amazonaws.com.amazonaws.secretsmanager.sql.AWSSecretsManagerRedshiftDriv

// Set the endpoint and port. You can also retrieve it from a key/value pair in the
 secret.
String URL = "jdbc-secretsmanager:redshift://example.com:5439";

// Populate the user property with the secret ARN to retrieve user and password from
 the secret
Properties info = new Properties( );
info.put( "user", "secretId" );

// Establish the connection
conn = DriverManager.getConnection(URL, info);
```

Use c3p0 connection pooling to establish a connection

The following example shows how to establish a connection pool with a `c3p0.properties` file that uses the driver to retrieve credentials and connection information from the secret. For `user` and `jdbcUrl`, enter the secret ID to configure the connection pool. Then you can retrieve connections from the pool and use them as any other database connections. For more information, see JDBC Basics on the Java documentation website.

For more information about c3p0, see c3p0 on the Machinery For Change website.

MySQL

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerMySQLDriver
c3p0.jdbcUrl=secretId
```

PostgreSQL

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerPostgreSQLDriver
c3p0.jdbcUrl=secretId
```

Oracle

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerOracleDriver
c3p0.jdbcUrl=secretId
```

MSSQLServer

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerMSSQLServerDriver
c3p0.jdbcUrl=secretId
```

Db2

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerDb2Driver
c3p0.jdbcUrl=secretId
```

Redshift

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerRedshiftDriver
c3p0.jdbcUrl=secretId
```

Use c3p0 connection pooling to establish a connection by specifying the endpoint and port

The following example shows how to establish a connection pool with a `c3p0.properties` file that uses the driver to retrieve credentials in a secret with an endpoint and port that you specify. Then you can retrieve connections from the pool and use them as any other database connections. For more information, see JDBC Basics on the Java documentation website.

Amazon RDS managed secrets don't include the endpoint and port of the database. To connect to a database using master credentials in a secret that's managed by Amazon RDS, you specify them in your code.

Secrets that are replicated to other Regions can improve latency for the connection to the regional database, but they do not contain different connection information from the source secret. Each replica is a copy of the source secret. To store regional connection information in the secret, add more key/value pairs for the endpoint and port information for the Regions.

MySQL

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerMySQLDriver
c3p0.jdbcUrl=jdbc-secretsmanager:mysql://example.com:3306
```

PostgreSQL

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerPostgreSQLDriver
c3p0.jdbcUrl=jdbc-secretsmanager:postgresql://example.com:5432/database
```

Oracle

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerOracleDriver
c3p0.jdbcUrl=jdbc-secretsmanager:oracle:thin:@example.com:1521/ORCL
```

MSSQLServer

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerMSSQLServerDriver
c3p0.jdbcUrl=jdbc-secretsmanager:sqlserver://example.com:1433
```

Db2

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerDb2Driver
c3p0.jdbcUrl=jdbc-secretsmanager:db2://example.com:50000
```

Redshift

```
c3p0.user=secretId
c3p0.driverClass=com.amazonaws.secretsmanager.sql.AWSSecretsManagerRedshiftDriver
c3p0.jdbcUrl=jdbc-secretsmanager:redshift://example.com:5439
```

Get a Secrets Manager secret value using the Java AWS SDK

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

- If you store database credentials in the secret, use the Secrets Manager SQL connection drivers to connect to a database using the credentials in the secret.
- For other types of secrets, use the Secrets Manager Java-based caching component or call the SDK directly with `GetSecretValue` or `BatchGetSecretValue`.

The following code examples show how to use `GetSecretValue`.

Required permissions: `secretsmanager:GetSecretValue`

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.secretsmanager.SecretsManagerClient;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueRequest;
import software.amazon.awssdk.services.secretsmanager.model.GetSecretValueResponse;
import software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
```

```
 *
 * We recommend that you cache your secret values by using client-side caching.
 *
 * Caching secrets improves speed and reduces your costs. For more information,
 * see the following documentation topic:
 *
 * https://docs.aws.amazon.com/secretsmanager/latest/userguide/retrieving-secrets.html
 */
public class GetSecretValue {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <secretName>\s

                Where:
                    secretName - The name of the secret (for example, tutorials/
MyFirstSecret).\s
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String secretName = args[0];
        Region region = Region.US_EAST_1;
        SecretsManagerClient secretsClient = SecretsManagerClient.builder()
                .region(region)
                .build();

        getValue(secretsClient, secretName);
        secretsClient.close();
    }

    public static void getValue(SecretsManagerClient secretsClient, String secretName)
 {
        try {
            GetSecretValueRequest valueRequest = GetSecretValueRequest.builder()
                    .secretId(secretName)
                    .build();

            GetSecretValueResponse valueResponse =
 secretsClient.getSecretValue(valueRequest);
```

```
            String secret = valueResponse.secretString();
            System.out.println(secret);

        } catch (SecretsManagerException e) {
            System.err.println(e.awsErrorDetails().errorMessage());
            System.exit(1);
        }
    }
}
```

Get a Secrets Manager secret value using Python

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

Topics

Get a Secrets Manager secret value using Python with client-side caching

When you retrieve a secret, you can use the Secrets Manager Python-based caching component to cache it for future use. Retrieving a cached secret is faster than retrieving it from Secrets Manager. Because there is a cost for calling Secrets Manager APIs, using a cache can reduce your costs. For all of the ways you can retrieve secrets, see Get secrets.

The cache policy is Least Recently Used (LRU), so when the cache must discard a secret, it discards the least recently used secret. By default, the cache refreshes secrets every hour. You can configure how often the secret is refreshed in the cache, and you can hook into the secret retrieval to add more functionality.

The cache does not force garbage collection once cache references are freed. The cache implementation does not include cache invalidation. The cache implementation is focused around the cache itself, and is not security hardened or focused. If you require additional security such as encrypting items in the cache, use the interfaces and abstract methods provided.

To use the component, you must have the following:

- Python 3.6 or later.
- botocore 1.12 or higher. See AWS SDK for Python and Botocore.
- setuptools_scm 3.2 or higher. See https://pypi.org/project/setuptools-scm/.

To download the source code, see Secrets Manager Python-based caching client component on GitHub.

To install the component, use the following command.

```
$ pip install aws-secretsmanager-caching
```

Required permissions:

- `secretsmanager:DescribeSecret`
- `secretsmanager:GetSecretValue`

For more information, see Permissions reference.

Reference

- SecretCache
- SecretCacheConfig
- SecretCacheHook
- @InjectSecretString
- @InjectKeywordedSecretString

Example Retrieve a secret

The following example shows how to get the secret value for a secret named *mysecret*.

```
import botocore
import botocore.session
from aws_secretsmanager_caching import SecretCache, SecretCacheConfig

client = botocore.session.get_session().create_client('secretsmanager')
cache_config = SecretCacheConfig()
```

```
cache = SecretCache( config = cache_config, client = client)

secret = cache.get_secret_string('mysecret')
```

SecretCache

An in-memory cache for secrets retrieved from Secrets Manager. You use the section called "get_secret_string" or the section called "get_secret_binary" to retrieve a secret from the cache. You can configure the cache settings by passing in a the section called "SecretCacheConfig" object in the constructor.

For more information, including examples, see the section called "Python with client-side caching".

```
cache = SecretCache(
    config = the section called "SecretCacheConfig",
    client = client
)
```

These are the available methods:

- get_secret_string
- get_secret_binary

get_secret_string

Retrieves the secret string value.

Request syntax

```
response = cache.get_secret_string(
    secret_id='string',
    version_stage='string' )
```

Parameters

- `secret_id` (*string*): [Required] The name or ARN of the secret.
- `version_stage` (*string*): The version of secrets that you want to retrieve. For more information, see secret versions. The default is 'AWSCURRENT'.

Return type

string

get_secret_binary

Retrieves the secret binary value.

Request syntax

```
response = cache.get_secret_binary(
    secret_id='string',
    version_stage='string'
)
```

Parameters

- `secret_id` (*string*): [Required] The name or ARN of the secret.
- `version_stage` (*string*): The version of secrets that you want to retrieve. For more information, see secret versions. The default is 'AWSCURRENT'.

Return type

base64-encoded string

SecretCacheConfig

Cache configuration options for a the section called "SecretCache" such as max cache size and Time to Live (TTL) for cached secrets.

Parameters

`max_cache_size` (*int*)

The maximum cache size. The default is `1024` secrets.

`exception_retry_delay_base` (*int*)

The number of seconds to wait after an exception is encountered before retrying the request. The default is 1.

`exception_retry_growth_factor` (*int*)pur

The growth factor to use for calculating the wait time between retries of failed requests. The default is 2.

`exception_retry_delay_max` (*int*)

The maximum amount of time in seconds to wait between failed requests. The default is `3600`.

`default_version_stage` (*str*)

The version of secrets that you want to cache. For more information, see Secret versions. The default is `'AWSCURRENT'`.

`secret_refresh_interval` (*int*)

The number of seconds to wait between refreshing cached secret information. The default is `3600`.

`secret_cache_hook` (*SecretCacheHook*)

An implementation of the `SecretCacheHook` abstract class. The default value is `None`.

SecretCacheHook

An interface to hook into a the section called "SecretCache" to perform actions on the secrets being stored in the cache.

These are the available methods:

- put
- get

put

Prepares the object for storing in the cache.

Request syntax

```
response = hook.put(
    obj='secret_object'
)
```

Parameters

- `obj` (*object*) -- [Required] The secret or object that contains the secret.

Return type

object

get

Derives the object from the cached object.

Request syntax

```
response = hook.get(
    obj='secret_object'
)
```

Parameters

- `obj` (*object*): [Required] The secret or object that contains the secret.

Return type

object

@InjectSecretString

This decorator expects a secret ID string and the section called "SecretCache" as the first and second arguments. The decorator returns the secret string value. The secret must contain a string.

```
from aws_secretsmanager_caching import SecretCache
from aws_secretsmanager_caching import InjectKeywordedSecretString,
 InjectSecretString

cache = SecretCache()

@InjectSecretString ( 'mysecret' ,  cache )
def function_to_be_decorated( arg1,  arg2,  arg3):
```

@InjectKeywordedSecretString

This decorator expects a secret ID string and the section called "SecretCache" as the first and second arguments. The remaining arguments map parameters from the wrapped function to JSON keys in the secret. The secret must contain a string in JSON structure.

For a secret that contains this JSON:

```
{
  "username": "saanvi",
  "password": "EXAMPLE-PASSWORD"
```

```
}
```

The following example shows how to extract the JSON values for `username` and `password` from the secret.

```
from aws_secretsmanager_caching import SecretCache
  from aws_secretsmanager_caching import InjectKeywordedSecretString,
 InjectSecretString

  cache = SecretCache()

  @InjectKeywordedSecretString ( secret_id = 'mysecret' ,  cache = cache ,
 func_username = 'username' ,  func_password = 'password' )
  def function_to_be_decorated( func_username,  func_password):
       print( 'Do something with the func_username and func_password parameters')
```

Get a Secrets Manager secret value using the Python AWS SDK

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

For Python applications, use the Secrets Manager Python-based caching component or call the SDK directly with `get_secret_value` or `batch_get_secret_value`.

The following code examples show how to use `GetSecretValue`.

Required permissions: `secretsmanager:GetSecretValue`

```
"""
Purpose

Shows how to use the AWS SDK for Python (Boto3) with AWS
Secrets Manager to get a specific of secrets that match a
specified name
"""
import boto3
import logging

from get_secret_value import GetSecretWrapper

# Configure logging
```

```
logging.basicConfig(level=logging.INFO)


def run_scenario(secret_name):
    """
    Retrieve a secret from AWS Secrets Manager.

    :param secret_name: Name of the secret to retrieve.
    :type secret_name: str
    """
    try:
        # Validate secret_name
        if not secret_name:
            raise ValueError("Secret name must be provided.")
        # Retrieve the secret by name
        client = boto3.client("secretsmanager")
        wrapper = GetSecretWrapper(client)
        secret = wrapper.get_secret(secret_name)
        # Note: Secrets should not be logged.
        return secret
    except Exception as e:
        logging.error(f"Error retrieving secret: {e}")
        raise

class GetSecretWrapper:
    def __init__(self, secretsmanager_client):
        self.client = secretsmanager_client


    def get_secret(self, secret_name):
        """
        Retrieve individual secrets from AWS Secrets Manager using the get_secret_value
 API.
        This function assumes the stack mentioned in the source code README has been
 successfully deployed.
        This stack includes 7 secrets, all of which have names beginning with
 "mySecret".

        :param secret_name: The name of the secret fetched.
        :type secret_name: str
        """
        try:
            get_secret_value_response = self.client.get_secret_value(
                SecretId=secret_name
```

```
            )
            logging.info("Secret retrieved successfully.")
            return get_secret_value_response["SecretString"]
        except self.client.exceptions.ResourceNotFoundException:
            msg = f"The requested secret {secret_name} was not found."
            logger.info(msg)
            return msg
        except Exception as e:
            logger.error(f"An unknown error occurred: {str(e)}.")
            raise
```

Get a batch of Secrets Manager secret values using the Python AWS SDK

The following code example shows how to get a batch of Secrets Manager secret values.

Required permissions:

- `secretsmanager:BatchGetSecretValue`
- `secretsmanager:GetSecretValue` permission for each secret you want to retrieve.
- If you use filters, you must also have `secretsmanager:ListSecrets`.

For an example permissions policy, see the section called “Example: Permission to retrieve a group of secret values in a batch”.

> **Important**
>
> If you have a VPCE policy that denies permission to retrieve an individual secret in the group you are retrieving, `BatchGetSecretValue` will not return any secret values, and it will return an error.

```
class BatchGetSecretsWrapper:
    def __init__(self, secretsmanager_client):
        self.client = secretsmanager_client
```

```
    def batch_get_secrets(self, filter_name):
        """
        Retrieve multiple secrets from AWS Secrets Manager using the
batch_get_secret_value API.
        This function assumes the stack mentioned in the source code README has been
successfully deployed.
        This stack includes 7 secrets, all of which have names beginning with
"mySecret".

        :param filter_name: The full or partial name of secrets to be fetched.
        :type filter_name: str
        """
        try:
            secrets = []
            response = self.client.batch_get_secret_value(
                Filters=[{"Key": "name", "Values": [f"{filter_name}"]}]
            )
            for secret in response["SecretValues"]:
                secrets.append(json.loads(secret["SecretString"]))
            if secrets:
                logger.info("Secrets retrieved successfully.")
            else:
                logger.info("Zero secrets returned without error.")
            return secrets
        except self.client.exceptions.ResourceNotFoundException:
            msg = f"One or more requested secrets were not found with filter:
{filter_name}"
            logger.info(msg)
            return msg
        except Exception as e:
            logger.error(f"An unknown error occurred:\n{str(e)}.")
            raise
```

Get a Secrets Manager secret value using .NET

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

Topics

- Get a Secrets Manager secret value using .NET with client-side caching
- Get a Secrets Manager secret value using the SDK for .NET

Get a Secrets Manager secret value using .NET with client-side caching

When you retrieve a secret, you can use the Secrets Manager .NET-based caching component to cache it for future use. Retrieving a cached secret is faster than retrieving it from Secrets Manager. Because there is a cost for calling Secrets Manager APIs, using a cache can reduce your costs. For all of the ways you can retrieve secrets, see Get secrets.

The cache policy is Least Recently Used (LRU), so when the cache must discard a secret, it discards the least recently used secret. By default, the cache refreshes secrets every hour. You can configure how often the secret is refreshed in the cache, and you can hook into the secret retrieval to add more functionality.

The cache does not force garbage collection once cache references are freed. The cache implementation does not include cache invalidation. The cache implementation is focused around the cache itself, and is not security hardened or focused. If you require additional security such as encrypting items in the cache, use the interfaces and abstract methods provided.

To use the component, you must have the following:

- .NET Framework 4.6.2 or higher, or .NET Standard 2.0 or higher. See Download .NET on the Microsoft .NET website.
- The AWS SDK for .NET. See the section called “AWS SDKs”.

To download the source code, see Caching client for .NET on GitHub.

To use the cache, first instantiate it, then retrieve your secret by using `GetSecretString` or `GetSecretBinary`. On successive retrievals, the cache returns the cached copy of the secret.

To get the caching package

- Do one of the following:
  - Run the following .NET CLI command in your project directory.

    ```
    dotnet add package AWSSDK.SecretsManager.Caching --version 1.0.6
    ```

- Add the following package reference to your `.csproj` file.

```
<ItemGroup>
    <PackageReference Include="AWSSDK.SecretsManager.Caching" Version="1.0.6" /
>
</ItemGroup>
```

Required permissions:

- `secretsmanager:DescribeSecret`
- `secretsmanager:GetSecretValue`

For more information, see Permissions reference.

Reference

- SecretsManagerCache
- SecretCacheConfiguration
- ISecretCacheHook

Example Retrieve a secret

The following code example shows a method that retrieves a secret named *MySecret*.

```
using Amazon.SecretsManager.Extensions.Caching;

namespace LambdaExample
{
    public class CachingExample
    {
        private const string MySecretName ="MySecret";

        private SecretsManagerCache cache = new SecretsManagerCache();

        public async Task<Response>  FunctionHandlerAsync(string input, ILambdaContext
 context)
        {
            string MySecret = await cache.GetSecretString(MySecretName);

            // Use the secret, return success
```

```
        }
    }
}
```

Example Configure the time to live (TTL) cache refresh duration

The following code example shows a method that retrieves a secret named *MySecret* and sets the TTL cache refresh duration to 24 hours.

```
using Amazon.SecretsManager.Extensions.Caching;

namespace LambdaExample
{
    public class CachingExample
    {
        private const string MySecretName = "MySecret";

        private static SecretCacheConfiguration cacheConfiguration = new
 SecretCacheConfiguration
        {
            CacheItemTTL = 86400000
        };
        private SecretsManagerCache cache = new
 SecretsManagerCache(cacheConfiguration);
        public async Task<Response> FunctionHandlerAsync(string input, ILambdaContext
 context)
        {
            string mySecret = await cache.GetSecretString(MySecretName);

            // Use the secret, return success
        }
    }
}
```

SecretsManagerCache

An in-memory cache for secrets requested from Secrets Manager. You use the section called "GetSecretString" or the section called "GetSecretBinary" to retrieve a secret from the cache. You can configure the cache settings by passing in a the section called "SecretCacheConfiguration" object in the constructor.

For more information, including examples, see the section called ".NET with client-side caching".

Constructors

`public SecretsManagerCache()`

Default constructor for a `SecretsManagerCache` object.

`public SecretsManagerCache(IAmazonSecretsManager secretsManager)`

Constructs a new cache using a Secrets Manager client created using the provided AmazonSecretsManagerClient. Use this constructor to customize the Secrets Manager client, for example to use a specific region or endpoint.

Parameters

secretsManager

The AmazonSecretsManagerClient to retrieve secrets from.

`public SecretsManagerCache(SecretCacheConfiguration config)`

Constructs a new secret cache using the provided the section called "SecretCacheConfiguration". Use this constructor to configure the cache, for example the number of secrets to cache and how often it refreshes.

Parameters

config

A the section called "SecretCacheConfiguration" that contains configuration information for the cache.

`public SecretsManagerCache(IAmazonSecretsManager secretsManager, SecretCacheConfiguration config)`

Constructs a new cache using a Secrets Manager client created using the provided AmazonSecretsManagerClient and a the section called "SecretCacheConfiguration". Use this constructor to customize the Secrets Manager client, for example to use a specific region or endpoint as well as configure the cache, for example the number of secrets to cache and how often it refreshes.

Parameters

secretsManager

The [AmazonSecretsManagerClient] to retrieve secrets from.

config

A [the section called “SecretCacheConfiguration”] that contains configuration information for the cache.

Methods

GetSecretString

```
public async Task<String> GetSecretString(String secretId)
```

Retrieves a string secret from Secrets Manager.

Parameters

secretId

The ARN or name of the secret to retrieve.

GetSecretBinary

```
public async Task<byte[]> GetSecretBinary(String secretId)
```

Retrieves a binary secret from Secrets Manager.

Parameters

secretId

The ARN or name of the secret to retrieve.

RefreshNowAsync

```
public async Task<bool> RefreshNowAsync(String secretId)
```

Requests the secret value from Secrets Manager and updates the cache with any changes. If there is no existing cache entry, creates a new one. Returns `true` if the refresh is successful.

Parameters

secretId

The ARN or name of the secret to retrieve.

GetCachedSecret

```
public SecretCacheItem GetCachedSecret(string secretId)
```

Returns the cache entry for the specified secret if it exists in the cache. Otherwise, retrieves the secret from Secrets Manager and creates a new cache entry.

Parameters

secretId

The ARN or name of the secret to retrieve.

SecretCacheConfiguration

Cache configuration options for a the section called “SecretsManagerCache”, such as maximum cache size and Time to Live (TTL) for cached secrets.

Properties

CacheItemTTL

```
public uint CacheItemTTL { get; set; }
```

The TTL of a cache item in milliseconds. The default is `3600000` ms or 1 hour. The maximum is `4294967295` ms, which is approximately 49.7 days.

MaxCacheSize

```
public ushort MaxCacheSize { get; set; }
```

The maximum cache size. The default is 1024 secrets. The maximum is 65,535.

VersionStage

```
public string VersionStage { get; set; }
```

The version of secrets that you want to cache. For more information, see Secret versions. The default is "AWSCURRENT".

Client

```
public IAmazonSecretsManager Client { get; set; }
```

The AmazonSecretsManagerClient to retrieve secrets from. If it is `null`, the cache instantiates a new client. The default is `null`.

CacheHook

```
public ISecretCacheHook CacheHook { get; set; }
```

A the section called "ISecretCacheHook".

ISecretCacheHook

An interface to hook into a the section called "SecretsManagerCache" to perform actions on the secrets being stored in the cache.

Methods

Put

```
object Put(object o);
```

Prepare the object for storing in the cache.

Returns the object to store in the cache.

Get

```
object Get(object cachedObject);
```

Derive the object from the cached object.

Returns the object to return from the cache

Get a Secrets Manager secret value using the SDK for .NET

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

For .NET applications, use the Secrets Manager .NET-based caching component or call the SDK directly with `GetSecretValue` or `BatchGetSecretValue`.

The following code examples show how to use `GetSecretValue`.

Required permissions: `secretsmanager:GetSecretValue`

```
using System;
using System.IO;
using System.Threading.Tasks;
using Amazon.SecretsManager;
using Amazon.SecretsManager.Model;

/// <summary>
/// This example uses the Amazon Web Service Secrets Manager to retrieve
/// the secret value for the provided secret name.
/// </summary>
public class GetSecretValue
{
    /// <summary>
    /// The main method initializes the necessary values and then calls
    /// the GetSecretAsync and DecodeString methods to get the decoded
    /// secret value for the secret named in secretName.
    /// </summary>
    public static async Task Main()
    {
        string secretName = "<<{{MySecretName}}>>";
        string secret;

        IAmazonSecretsManager client = new AmazonSecretsManagerClient();

        var response = await GetSecretAsync(client, secretName);

        if (response is not null)
        {
            secret = DecodeString(response);

            if (!string.IsNullOrEmpty(secret))
            {
                Console.WriteLine($"The decoded secret value is: {secret}.");
            }
            else
            {
                Console.WriteLine("No secret value was returned.");
```

```
                }
            }
        }

        /// <summary>
        /// Retrieves the secret value given the name of the secret to
        /// retrieve.
        /// </summary>
        /// <param name="client">The client object used to retrieve the secret
        /// value for the given secret name.</param>
        /// <param name="secretName">The name of the secret value to retrieve.</param>
        /// <returns>The GetSecretValueReponse object returned by
        /// GetSecretValueAsync.</returns>
        public static async Task<GetSecretValueResponse> GetSecretAsync(
            IAmazonSecretsManager client,
            string secretName)
        {
            GetSecretValueRequest request = new GetSecretValueRequest()
            {
                SecretId = secretName,
                VersionStage = "AWSCURRENT", // VersionStage defaults to AWSCURRENT if
 unspecified.
            };

            GetSecretValueResponse response = null;

            // For the sake of simplicity, this example handles only the most
            // general SecretsManager exception.
            try
            {
                response = await client.GetSecretValueAsync(request);
            }
            catch (AmazonSecretsManagerException e)
            {
                Console.WriteLine($"Error: {e.Message}");
            }

            return response;
        }

        /// <summary>
        /// Decodes the secret returned by the call to GetSecretValueAsync and
        /// returns it to the calling program.
        /// </summary>
```

```
        /// <param name="response">A GetSecretValueResponse object containing
        /// the requested secret value returned by GetSecretValueAsync.</param>
        /// <returns>A string representing the decoded secret value.</returns>
        public static string DecodeString(GetSecretValueResponse response)
        {
            // Decrypts secret using the associated AWS Key Management Service
            // Customer Master Key (CMK.) Depending on whether the secret is a
            // string or binary value, one of these fields will be populated.
            if (response.SecretString is not null)
            {
                var secret = response.SecretString;
                return secret;
            }
            else if (response.SecretBinary is not null)
            {
                var memoryStream = response.SecretBinary;
                StreamReader reader = new StreamReader(memoryStream);
                string decodedBinarySecret =
  System.Text.Encoding.UTF8.GetString(Convert.FromBase64String(reader.ReadToEnd()));
                return decodedBinarySecret;
            }
            else
            {
                return string.Empty;
            }
        }
    }
```

Get a Secrets Manager secret value using Go

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

Topics

Get a Secrets Manager secret value using Go with client-side caching

When you retrieve a secret, you can use the Secrets Manager Go-based caching component to cache it for future use. Retrieving a cached secret is faster than retrieving it from Secrets Manager. Because there is a cost for calling Secrets Manager APIs, using a cache can reduce your costs. For all of the ways you can retrieve secrets, see Get secrets.

The cache policy is Least Recently Used (LRU), so when the cache must discard a secret, it discards the least recently used secret. By default, the cache refreshes secrets every hour. You can configure how often the secret is refreshed in the cache, and you can hook into the secret retrieval to add more functionality.

The cache does not force garbage collection once cache references are freed. The cache implementation does not include cache invalidation. The cache implementation is focused around the cache itself, and is not security hardened or focused. If you require additional security such as encrypting items in the cache, use the interfaces and abstract methods provided.

To use the component, you must have the following:

- AWS SDK for Go. See the section called “AWS SDKs”.

To download the source code, see Secrets Manager Go caching client on GitHub.

To set up a Go development environment, see Golang Getting Started on the Go Programming Language website.

Required permissions:

- `secretsmanager:DescribeSecret`
- `secretsmanager:GetSecretValue`

For more information, see Permissions reference.

Reference

- type Cache
- type CacheConfig
- type CacheHook

Example Retrieve a secret

The following code example shows a Lambda function that retrieves a secret.

```
package main

import (
  "github.com/aws/aws-lambda-go/lambda"
  "github.com/aws/aws-secretsmanager-caching-go/secretcache"
)

var (
  secretCache, _ = secretcache.New()
)

func HandleRequest(secretId string) string {
  result, _ := secretCache.GetSecretString(secretId)

  // Use the secret, return success
}

 func main() {
  lambda. Start( HandleRequest)
}
```

type Cache

An in-memory cache for secrets requested from Secrets Manager. You use the section called “GetSecretString” or the section called “GetSecretBinary” to retrieve a secret from the cache.

The following example shows how to configure the cache settings.

```
// Create a custom secretsmanager client
client := getCustomClient()

// Create a custom CacheConfig struct
config := secretcache. CacheConfig{
    MaxCacheSize:  secretcache.DefaultMaxCacheSize + 10,
    VersionStage:  secretcache.DefaultVersionStage,
    CacheItemTTL:  secretcache.DefaultCacheItemTTL,
}

// Instantiate the cache
```

```
cache, _ := secretcache.New(
    func( c *secretcache.Cache) {  c. CacheConfig = config },
    func( c *secretcache.Cache) {  c. Client = client },
)
```

For more information, including examples, see the section called “Go with client-side caching”.

Methods

New

`func New(optFns ...func(*Cache)) (*Cache, error)`

New constructs a secret cache using functional options, uses defaults otherwise. Initializes a SecretsManager Client from a new session. Initializes CacheConfig to default values. Initialises LRU cache with a default max size.

GetSecretString

`func (c *Cache) GetSecretString(secretId string) (string, error)`

GetSecretString gets the secret string value from the cache for given secret ID. Returns the secret string and an error if operation failed.

GetSecretStringWithStage

`func (c *Cache) GetSecretStringWithStage(secretId string, versionStage string) (string, error)`

GetSecretStringWithStage gets the secret string value from the cache for given secret ID and version stage. Returns the secret string and an error if operation failed.

GetSecretBinary

`func (c *Cache) GetSecretBinary(secretId string) ([]byte, error) {`

GetSecretBinary gets the secret binary value from the cache for given secret ID. Returns the secret binary and an error if operation failed.

GetSecretBinaryWithStage

`func (c *Cache) GetSecretBinaryWithStage(secretId string, versionStage string) ([]byte, error)`

GetSecretBinaryWithStage gets the secret binary value from the cache for given secret ID and version stage. Returns the secret binary and an error if operation failed.

type CacheConfig

Cache configuration options for a Cache, such as maximum cache size, default version stage, and Time to Live (TTL) for cached secrets.

```
type CacheConfig struct {

    // The maximum cache size. The default is 1024 secrets.
    MaxCacheSize int

    // The TTL of a cache item in nanoseconds. The default is
    // 3.6e10^12 ns or 1 hour.
    CacheItemTTL int64

    // The version of secrets that you want to cache. The default
    // is "AWSCURRENT".
    VersionStage string

    // Used to hook in-memory cache updates.
    Hook CacheHook
}
```

type CacheHook

An interface to hook into a Cache to perform actions on the secret being stored in the cache.

Methods

Put

`Put(data interface{}) interface{}`

Prepares the object for storing in the cache.

Get

`Get(data interface{}) interface{}`

Derives the object from the cached object.

Get a Secrets Manager secret value using the Go AWS SDK

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

For Go applications, use the Secrets Manager Go-based caching component or call the SDK directly with `GetSecretValue` or `BatchGetSecretValue`.

The following code example shows how to get a Secrets Manager secret value.

Required permissions: `secretsmanager:GetSecretValue`

```
// Use this snippet in your app.
// If you need more information about configurations or implementing the sample code,
visit the AWS docs:
// https://aws.github.io/aws-sdk-go-v2/docs/getting-started/

import (
  "context"
  "log"

  "github.com/aws/aws-sdk-go-v2/aws"
  "github.com/aws/aws-sdk-go-v2/config"
  "github.com/aws/aws-sdk-go-v2/service/secretsmanager"
)

func main() {
  secretName := "<<{{MySecretName}}>>"
  region := "<<{{MyRegionName}}>>"

  config, err := config.LoadDefaultConfig(context.TODO(), config.WithRegion(region))
  if err != nil {
    log.Fatal(err)
  }

  // Create Secrets Manager client
  svc := secretsmanager.NewFromConfig(config)

  input := &secretsmanager.GetSecretValueInput{
    SecretId:     aws.String(secretName),
    VersionStage: aws.String("AWSCURRENT"), // VersionStage defaults to AWSCURRENT if
unspecified
```

```
    }

    result, err := svc.GetSecretValue(context.TODO(), input)
    if err != nil {
      // For a list of exceptions thrown, see
      // https://<<{{DocsDomain}}>>/secretsmanager/latest/apireference/
API_GetSecretValue.html
      log.Fatal(err.Error())
    }

    // Decrypts secret using the associated KMS key.
    var secretString string = *result.SecretString

    // Your code goes here.
  }
```

Get a Secrets Manager secret value using Rust

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

Topics

- Get a Secrets Manager secret value using Rust with client-side caching
- Get a Secrets Manager secret value using the Rust AWS SDK

Get a Secrets Manager secret value using Rust with client-side caching

When you retrieve a secret, you can use the Secrets Manager Rust-based caching component to cache it for future use. Retrieving a cached secret is faster than retrieving it from Secrets Manager. Because there is a cost for calling Secrets Manager APIs, using a cache can reduce your costs. For all of the ways you can retrieve secrets, see Get secrets.

The cache policy is First In First Out (FIFO), so when the cache must discard a secret, it discards the oldest secret. By default, the cache refreshes secrets every hour. You can configure the following:

- `max_size` – The maximum number of cached secrets to maintain before evicting secrets that have not been accessed recently.

- `ttl` – The duration a cached item is considered valid before requiring a refresh of the secret state.

The cache implementation does not include cache invalidation. The cache implementation is focused around the cache itself, and is not security hardened or focused. If you require additional security such as encrypting items in the cache, use the traits provided to modify the cache.

To use the component, you must have a Rust 2021 development environment with `tokio`. For more information, see Getting started on the Rust Programming Language website.

To download the source code, see Secrets Manager Rust-based caching client component on GitHub.

To install the caching component, use the following command.

```
cargo add aws_secretsmanager_caching
```

Required permissions:

- `secretsmanager:DescribeSecret`
- `secretsmanager:GetSecretValue`

For more information, see Permissions reference.

Example Retrieve a secret

The following example shows how to get the secret value for a secret named *MyTest*.

```
use aws_secretsmanager_caching::SecretsManagerCachingClient;
use std::num::NonZeroUsize;
use std::time::Duration;

let client = match SecretsManagerCachingClient::default(
    NonZeroUsize::new(10).unwrap(),
    Duration::from_secs(60),
)
.await
{
    Ok(c) => c,
    Err(_) => panic!("Handle this error"),
```

```
};

let secret_string = match client.get_secret_value("MyTest", None, None).await {
    Ok(s) => s.secret_string.unwrap(),
    Err(_) => panic!("Handle this error"),
};

// Your code here
```

Example Instantiating Cache with a custom configuration and a custom client

The following example shows how to configure the cache and then get the secret value for a secret named *MyTest*.

```
let config = aws_config::load_defaults(BehaviorVersion::latest())
    .await
    .into_builder()
    .region(Region::from_static("us-west-2"))
    .build();

let asm_builder = aws_sdk_secretsmanager::config::Builder::from(&config);

let client = match SecretsManagerCachingClient::from_builder(
    asm_builder,
    NonZeroUsize::new(10).unwrap(),
    Duration::from_secs(60),
)
.await
{
    Ok(c) => c,
    Err(_) => panic!("Handle this error"),
};

let secret_string = client
    .get_secret_value("MyTest", None, None)
    .await
    {
        Ok(c) => c.secret_string.unwrap(),
        Err(_) => panic!("Handle this error"),
    };

// Your code here
```
````
````

Get a Secrets Manager secret value using the Rust AWS SDK

In applications, you can retrieve your secrets by calling `GetSecretValue` or `BatchGetSecretValue`in any of the AWS SDKs. However, we recommend that you cache your secret values by using client-side caching. Caching secrets improves speed and reduces your costs.

For Rust applications, use the Secrets Manager Rust-based caching component or call the SDK directly with GetSecretValue or BatchGetSecretValue.

The following code example shows how to get a Secrets Manager secret value.

Required permissions: `secretsmanager:GetSecretValue`

```
async fn show_secret(client: &Client, name: &str) -> Result<(), Error> {
    let resp = client.get_secret_value().secret_id(name).send().await?;

    println!("Value: {}", resp.secret_string().unwrap_or("No value!"));

    Ok(())
}
```

Use AWS Secrets Manager secrets in Amazon Elastic Kubernetes Service

To show secrets from AWS Secrets Manager (ASCP) as files mounted in Amazon EKS Pods, you can use the AWS Secrets and Configuration Provider for the Kubernetes Secrets Store CSI Driver. The ASCP works with Amazon Elastic Kubernetes Service 1.17+ running an Amazon EC2 node group. AWS Fargate node groups are not supported. With the ASCP, you can store and manage your secrets in Secrets Manager and then retrieve them through your workloads running on Amazon EKS. If your secret contains multiple key-value pairs in JSON format, you can choose which ones to mount in Amazon EKS. The ASCP uses JMESPath syntax to query the key-value pairs in your secret. The ASCP also works with Parameter Store parameters. The ASCP offers two methods of authentication with Amazon EKS The first approach uses IAM Roles for Service Accounts (IRSA). The second approach uses Pod Identities. Each approach has its benefits and use cases.

ASCP with IAM Roles for Service Accounts (IRSA)

The ASCP with IAM Roles for Service Accounts (IRSA) allows you to mount secrets from AWS Secrets Manager as files in your Amazon EKS Pods. This approach is suitable when:

- You need to mount secrets as files in your Pods.
- You're using Amazon EKS version 1.17 or later with Amazon EC2 node groups.
- You want to retrieve specific key-value pairs from JSON-formatted secrets.

For more information, see the section called "Integrate ASCP with IRSA for Amazon EKS".

ASCP with Pod Identity

ASCP with EKS Pod Identity

The ASCP with Pod Identity method enhances security and simplifies configuration for accessing secrets in Amazon EKS. This approach is beneficial when:

- You need more granular permission management at the Pod level.
- You're using Amazon EKS version 1.24 or later.
- You want improved performance and scalability.

For more information, see the section called "Integrate ASCP with Pod Identity for Amazon EKS".

Choosing the right approach

Consider the following factors when deciding between ASCP with IRSA and ASCP with Pod Identity:

- Amazon EKSversion: Pod Identity requires Amazon EKS 1.24+, while CSI driver works with Amazon EKS 1.17+.
- Security requirements: Pod Identity offers more granular control at the Pod level.
- Performance: Pod Identity generally performs better in high-scale environments.
- Complexity: Pod Identity simplifies setup by eliminating the need for separate service accounts.

Choose the method that best aligns with your specific requirements and Amazon EKS environment.

Install ASCP for Amazon EKS

This section explains how to install the AWS Secrets and Configuration Provider for Amazon EKS. With ASCP, you can mount secrets from Secrets Manager and parameters from AWS Systems Manager as files in Amazon EKS Pods.

Prerequisites

- An Amazon EKS cluster
  - Version 1.24 or later for Pod Identity
  - Version 1.17 or later for IRSA
- The AWS CLI installed and configured
- kubectl installed and configured for your Amazon EKS cluster
- Helm (version 3.0 or later)

Install and configure the ASCP

The ASCP is available on GitHub in the secrets-store-csi-provider-aws repository. The repo also contains example YAML files for creating and mounting a secret.

During installation, you can configure the ASCP to use a FIPS endpoint. For a list of endpoints, see the section called "Secrets Manager endpoints".

To install the ASCP as an EKS add-on

1. Install `eksctl` (installation instructions)
2. Run the following command to install the add-on with the default configuration:

   ```
   eksctl create addon --cluster <your_cluster> --name aws-secrets-store-csi-driver-
   provider
   ```

   If you'd like to configure the add-on, run the following installation command instead:

   ```
   aws eks create-addon --cluster-name <your_cluster> --addon-name aws-secrets-store-
   csi-driver-provider --configuration-values 'file://path/to/config.yaml'
   ```

   The configuration file can be a YAML or JSON file. To see the configuration schema for the add-on:

   a. Run the following command and note the latest version of the add-on:

      ```
      aws eks describe-addon-versions --addon-name aws-secrets-store-csi-driver-
      provider
      ```

b. Run the following command to see the add-on's configuration schema, replacing `<version>` with the version from the previous step:

```
aws eks describe-addon-configuration --addon-name aws-secrets-store-csi-driver-provider --addon-version <version>
```

To install the ASCP by using Helm

1. To make sure the repo is pointing to the latest charts, use `helm repo update.`
2. Install the chart. The following is an example of the `helm install` command:

```
helm install -n kube-system secrets-provider-aws aws-secrets-manager/secrets-store-csi-driver-provider-aws
```

a. To use a FIPS endpoint, add the following flag: `--set useFipsEndpoint=true`

b. To configure throttling, add the following flag: `--set-json 'k8sThrottlingParams={"qps": "`*`number of queries per second`*`", "burst": "`*`number of queries per second`*`"}'`

c. If the Secrets Store CSI Driver is already installed on your cluster, add the following flag: `--set secrets-store-csi-driver.install=false`. This will skip installing Secrets Store CSI Driver as a dependency.

To install by using the YAML in the repo

- Use the following commands.

```
helm repo add secrets-store-csi-driver https://kubernetes-sigs.github.io/secrets-store-csi-driver/charts
helm install -n kube-system csi-secrets-store secrets-store-csi-driver/secrets-store-csi-driver
kubectl apply -f https://raw.githubusercontent.com/aws/secrets-store-csi-driver-provider-aws/main/deployment/aws-provider-installer.yaml
```

Verify the installations

To verify the installations of your EKS cluster, Secrets Store CSI driver, and ASCP plugin, follow these steps:

1. Verify the EKS cluster:

   ```
   eksctl get cluster --name clusterName
   ```

   This command should return information about your cluster.

2. Verify the Secrets Store CSI driver installation:

   ```
   kubectl get pods -n kube-system -l app=secrets-store-csi-driver
   ```

   You should see Pods running with names like `csi-secrets-store-secrets-store-csi-driver-xxx`.

3. Verify the ASCP plugin installation:

   YAML installation

   ```
   $ kubectl get pods -n kube-system -l app=csi-secrets-store-provider-aws
   ```

   Example output:

   ```
   NAME                                    READY   STATUS    RESTARTS   AGE
   csi-secrets-store-provider-aws-12345     1/1     Running   0           2m
   ```

   Helm installation

   ```
   $  kubectl get pods -n kube-system -l app=secrets-store-csi-driver-provider-aws
   ```

   Example output:

   ```
   NAME                                                  READY   STATUS    RESTARTS
    AGE
   secrets-provider-aws-secrets-store-csi-driver-provider-67890       1/1
    Running   0          2m
   ```

You should see Pods in the `Running` state.

After running these commands, if everything is set up correctly, you should see all components running without any errors. If you encounter any issues, you may need to troubleshoot by checking the logs of the specific Pods that are having problems.

Troubleshooting

1. To check the logs of the ASCP provider, run:

```
kubectl logs -n kube-system -l app=csi-secrets-store-provider-aws
```

2. Check the status of all pods in the `kube-system` namespace:

```
kubectl -n kube-system get pods
```

```
kubectl -n kube-system logs pod/PODID
```

   All Pods related to the CSI driver and ASCP should be in the 'Running' state.

3. Check the CSI driver version:

```
kubectl get csidriver secrets-store.csi.k8s.io -o yaml
```

   This command should return information about the installed CSI driver.

Additional resources

For more information about using ASCP with Amazon EKS, see the following resources:

- Using Pod Identity with Amazon EKS
- AWS Secrets Store CSI Driver on GitHub

Use AWS Secrets and Configuration Provider CSI with Pod Identity for Amazon EKS

The AWS Secrets and Configuration Provider integration with the Pod Identity Agent for Amazon Elastic Kubernetes Service provides enhanced security, simplified configuration, and improved performance for applications running on Amazon EKS. Pod Identity simplifies IAM authentication for Amazon EKS when retrieving secrets from Secrets Manager or parameters from AWS Systems Manager Parameter Store.

Amazon EKS Pod Identity streamlines the process of configuring IAM permissions for Kubernetes applications by allowing permissions to be set up directly through Amazon EKS interfaces, reducing the number of steps and eliminating the need to switch between Amazon EKS and IAM services. Pod Identity enables the use of a single IAM role across multiple clusters without updating trust policies and supports role session tags for more granular access control. This approach not only simplifies policy management by allowing reuse of permission policies across roles but also enhances security by enabling access to AWS resources based on matching tags.

How it works

1. Pod Identity assigns an IAM role to the Pod.
2. ASCP uses this role to authenticate with AWS services.
3. If authorized, ASCP retrieves the requested secrets and makes them available to the Pod.

For more information, see Understand how Amazon EKS Pod Identity works in the *Amazon EKS User Guide*.

Prerequisites

> **Important**
>
> Pod Identity is supported only for Amazon EKS in the cloud. It is not supported for Amazon EKS Anywhere, Red Hat OpenShift Service on AWS, or self-managed Kubernetes clusters on Amazon EC2 instances.

- Amazon EKS cluster (version 1.24 or later)
- Access to AWS CLI and Amazon EKS cluster via `kubectl`

- Access to two AWS accounts (for cross-account access)

Install the Amazon EKS Pod Identity Agent

To use Pod Identity with your cluster, you must install the Amazon EKS Pod Identity Agent add-on.

To install the Pod Identity Agent

- Install the Pod Identity Agent add-on on your cluster:

```
eksctl create addon \
  --name eks-pod-identity-agent \
  --cluster clusterName \
  --region region
```

Set up ASCP with Pod Identity

1. Create a permissions policy that grants `secretsmanager:GetSecretValue` and `secretsmanager:DescribeSecret` permission to the secrets that the Pod needs to access. For an example policy, see the section called "Example: Permission to read and describe individual secrets".
2. Create an IAM role that can be assumed by the Amazon EKS service principal for Pod Identity:

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "Service": "pods.eks.amazonaws.com"
        },
        "Action": [
          "sts:AssumeRole",
          "sts:TagSession"
        ]
      }
    ]
```

```
    }
```

Attach the IAM policy to the role:

```
aws iam attach-role-policy \
  --role-name MY_ROLE \
  --policy-arn POLICY_ARN
```

3. Create a Pod Identity association. For an example, see [Create a Pod Identity association](#) in the *Amazon EKS User Guide*

4. Create the `SecretProviderClass` that specifies which secrets to mount in the Pod:

   ```
   kubectl apply -f https://raw.githubusercontent.com/aws/secrets-store-csi-driver-provider-aws/main/examples/ExampleSecretProviderClass-PodIdentity.yaml
   ```

   The key difference in `SecretProviderClass` between IRSA and Pod Identity is the optional parameter `usePodIdentity`. It is an optional field that determines the authentication approach. When not specified, it defaults to using IAM Roles for Service Accounts (IRSA).

   - To use EKS Pod Identity, use any of these values: `"true", "True", "TRUE", "t", "T"`.
   - To explicitly use IRSA, set to any of these values: `"false", "False", "FALSE", "f", or "F"`.

5. Deploy the Pod that mounts the secrets under `/mnt/secrets-store`:

   ```
   kubectl apply -f https://raw.githubusercontent.com/aws/secrets-store-csi-driver-provider-aws/main/examples/ExampleDeployment-PodIdentity.yaml
   ```

6. If you use a private Amazon EKS cluster, make sure that the VPC that the cluster is in has an AWS STS endpoint. For information about creating an endpoint, see [Interface VPC endpoints](#) in the *AWS Identity and Access Management User Guide*.

Verify the secret mount

To verify that the secret is mounted properly, run the following command:

```
kubectl exec -it $(kubectl get pods | awk '/pod-identity-deployment/{print $1}' | head
 -1) -- cat /mnt/secrets-store/MySecret
```

To set up Amazon EKS Pod Identity to access to secrets in Secrets Manager

1. Create a permissions policy that grants `secretsmanager:GetSecretValue` and `secretsmanager:DescribeSecret` permission to the secrets that the Pod needs to access. For an example policy, see the section called "Example: Permission to read and describe individual secrets".
2. Create a secret in Secrets Manager, if you do not already have one.

Troubleshoot

You can view most errors by describing the Pod deployment.

To see error messages for your container

1. Get a list of Pod names with the following command. If you aren't using the default namespace, use `-n` *`NAMESPACE`*.

```
kubectl get pods
```

2. To describe the Pod, in the following command, for *`PODID`* use the Pod ID from the Pods you found in the previous step. If you aren't using the default namespace, use `-n` *`NAMESPACE`*.

```
kubectl describe pod/PODID
```

To see errors for the ASCP

- To find more information in the provider logs, in the following command, for *`PODID`* use the ID of the *csi-secrets-store-provider-aws* Pod.

```
kubectl -n kube-system get pods
kubectl -n kube-system logs pod/PODID
```

Use AWS Secrets and Configuration Provider CSI with IAM Roles for Service Accounts (IRSA)

Topics

- Prerequisites
- Set up access control
- Identify which secrets to mount
- Troubleshoot

Prerequisites

- Amazon EKS cluster (version 1.17 or later)
- Access to AWS CLI and Amazon EKS cluster via `kubectl`

Set up access control

The ASCP retrieves the Amazon EKS Pod Identity and exchanges it for an IAM role. You set permissions in an IAM policy for that IAM role. When the ASCP assumes the IAM role, it gets access to the secrets you authorized. Other containers can't access the secrets unless you also associate them with the IAM role.

To grant your Amazon EKS Pod access to secrets in Secrets Manager

1. Create a permissions policy that grants `secretsmanager:GetSecretValue` and `secretsmanager:DescribeSecret` permission to the secrets that the Pod needs to access. For an example policy, see the section called “Example: Permission to read and describe individual secrets”.
2. Create an IAM OpenID Connect (OIDC) provider for the cluster if you don't already have one. For more information, see Create an IAM OIDC provider for your cluster in the *Amazon EKS User Guide*.
3. Create an IAM role for service account and attach the policy to it. For more information, see Create an IAM role for a service account in the *Amazon EKS User Guide*.
4. If you use a private Amazon EKS cluster, make sure that the VPC that the cluster is in has an AWS STS endpoint. For information about creating an endpoint, see Interface VPC endpoints in the *AWS Identity and Access Management User Guide*.

Identify which secrets to mount

To determine which secrets the ASCP mounts in Amazon EKS as files on the filesystem, you create a the section called "SecretProviderClass" YAML file. The `SecretProviderClass` lists the secrets to mount and the file name to mount them as. The `SecretProviderClass` must be in the same namespace as the Amazon EKS Pod it references.

Mount the secrets as files

The following instructions show how to mount secrets as files using example YAML files ExampleSecretProviderClass.yaml and ExampleDeployment.yaml.

To mount secrets in Amazon EKS

1. Apply the `SecretProviderClass` to the Pod:

   ```
   kubectl apply -f ExampleSecretProviderClass.yaml
   ```

2. Deploy your Pod:

   ```
   kubectl apply -f ExampleDeployment.yaml
   ```

3. The ASCP mounts the files.

Troubleshoot

You can view most errors by describing the Pod deployment.

To see error messages for your container

1. Get a list of Pod names with the following command. If you aren't using the default namespace, use `-n` *`nameSpace`*.

   ```
   kubectl get pods
   ```

2. To describe the Pod, in the following command, for *`podId`* use the Pod ID from the Pods you found in the previous step. If you aren't using the default namespace, use `-n` *`nameSpace`*.

   ```
   kubectl describe pod/podId
   ```

To see errors for the ASCP

- To find more information in the provider logs, in the following command, for *podId* use the ID of the *csi-secrets-store-provider-aws* Pod.

  ```
  kubectl -n kube-system get pods
  kubectl -n kube-system logs Pod/podId
  ```

- **Verify that the `SecretProviderClass` CRD is installed:**

  ```
  kubectl get crd secretproviderclasses.secrets-store.csi.x-k8s.io
  ```

  This command should return information about the `SecretProviderClass` custom resource definition.

- **Verify that the SecretProviderClass object was created.**

  ```
  kubectl get secretproviderclass SecretProviderClassName -o yaml
  ```

AWS Secrets and Configuration Provider code examples

ASCP authentication and access control examples

Example: IAM policy allowing Amazon EKS Pod Identity service (pods.eks.amazonaws.com) to assume the role and tag the session:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "pods.eks.amazonaws.com"
      },
      "Action": [
```

```
                "sts:AssumeRole",
                "sts:TagSession"
            ]
        }
    ]
}
```

SecretProviderClass

You use YAML to describe which secrets to mount in Amazon EKS using the ASCP. For examples, see the section called “SecretProviderClass usage”.

SecretProviderClass YAML structure

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
   name: name
spec:
  provider: aws
  parameters:
    region:
    failoverRegion:
    pathTranslation:
    usePodIdentity:
    preferredAddressType:
    objects:
```

The parameters field contains the details of the mount request:

region

(Optional) The AWS Region of the secret. If you don't use this field, the ASCP looks up the Region from the annotation on the node. This lookup adds overhead to mount requests, so we recommend that you provide the Region for clusters that use large numbers of Pods.

If you also specify `failoverRegion`, the ASCP tries to retrieve the secret from both Regions. If either Region returns a 4xx error, for example for an authentication issue, the ASCP does not mount either secret. If the secret is retrieved successfully from `region`, then the ASCP mounts that secret value. If the secret is not retrieved successfully from `region`, but it is retrieved successfully from `failoverRegion`, then the ASCP mounts that secret value.

failoverRegion

(Optional) If you include this field, the ASCP tries to retrieve the secret from the Regions defined in `region` and this field. If either Region returns a 4xx error, for example for an authentication issue, the ASCP does not mount either secret. If the secret is retrieved successfully from `region`, then the ASCP mounts that secret value. If the secret is not retrieved successfully from `region`, but it is retrieved successfully from `failoverRegion`, then the ASCP mounts that secret value. For an example of how to use this field, see Multi-Region secret failover.

pathTranslation

(Optional) A single substitution character to use if the file name in Amazon EKS will contain the path separator character, such as slash (/) on Linux. The ASCP can't create a mounted file that contains a path separator character. Instead, the ASCP replaces the path separator character with a different character. If you don't use this field, the replacement character is underscore (_), so for example, `My/Path/Secret` mounts as `My_Path_Secret`.

To prevent character substitution, enter the string `False`.

usePodIdentity

(Optional) Determines the authentication approach. When not specified, it defaults to IAM Roles for Service Accounts (IRSA) (IRSA).

- To use EKS Pod Identity, use any of these values: `"true""`, `"True"`, `"TRUE"`, `"t"`, or `"T"`.
- To explicitly use IRSA, set to any of these values: `"false"`, `"False"`, `"FALSE"`, `"f"`, or `"F""=`.

preferredAddressType

(Optional) Specifies the preferred IP address type for Pod Identity Agent endpoint communication. The field is only applicable when using EKS Pod Identity feature and will be ignored when using IAM Roles for Service Accounts.Values are case-insensitive. Valid values are:

- `"ipv4"`, `"IPv4""`, or `"IPV4"` – Force the use of Pod Identity Agent IPv4 endpoint
- `"ipv6"`, `"IPv6"`, or `"IPV6"` – Force the use of Pod Identity Agent IPv6 endpoint
- not specified – Use auto endpoint selection, trying IPv4 endpoint first and falling back to IPv6 endpoint if IPv4 fails

objects

A string containing a YAML declaration of the secrets to be mounted. We recommend using a YAML multi-line string or pipe (|) character.

objectName

Required. Specifies the name of the secret or parameter to be fetched. For Secrets Manager this is the [`SecretId`](#) parameter and can be either the friendly name or full ARN of the secret. For SSM Parameter Store, this is the [`Name`](#) of the parameter and can be either the name or full ARN of the parameter.

objectType

Required if you don't use a Secrets Manager ARN for `objectName`. Can be either `secretsmanager` or `ssmparameter`.

objectAlias

(Optional) The file name of the secret in the Amazon EKS Pod. If you don't specify this field, the `objectName` appears as the file name.

filePermission

(Optional) The 4 digit octal string which specifies the file permission to mount secret with. If you don't specify this field it will default to `"0644"`.

objectVersion

(Optional) The version ID of the secret. Not recommended because you must update the version ID every time you update the secret. By default the most recent version is used. If you include a `failoverRegion`, this field represents the primary `objectVersion`.

objectVersionLabel

(Optional) The alias for the version. The default is the most recent version AWSCURRENT. For more information, see [the section called "Secret versions"](#). If you include a `failoverRegion`, this field represents the primary `objectVersionLabel`.

jmesPath

(Optional) A map of the keys in the secret to the files to be mounted in Amazon EKS. To use this field, your secret value must be in JSON format. If you use this field, you must include the subfields `path` and `objectAlias`.

path

A key from a key-value pair in the JSON of the secret value. If the field contains a hyphen, use single quotes to escape it, for example: `path: '"hyphenated-path"'`

objectAlias

The file name to be mounted in the Amazon EKS Pod. If the field contains a hyphen, use single quotes to escape it, for example: `objectAlias: '"hyphenated-alias"'`

filePermission

(Optional) The 4 digit octal string which specifies the file permission to mount secret with. If you don't specify this field it will default to the parent object's file permission.

failoverObject

(Optional) If you specify this field, the ASCP tries to retrieve both the secret specified in the primary `objectName` and the secret specified in the `failoverObject` `objectName` sub-field. If either returns a 4xx error, for example for an authentication issue, the ASCP does not mount either secret. If the secret is retrieved successfully from the primary `objectName`, then the ASCP mounts that secret value. If the secret is not retrieved successfully from the primary `objectName`, but it is retrieved successfully from the failover `objectName`, then the ASCP mounts that secret value. If you include this field, you must include the field `objectAlias`. For an example of how to use this field, see Failover to a different secret.

You typically use this field when the failover secret isn't a replica. For an example of how to specify a replica, see Multi-Region secret failover.

objectName

The name or full ARN of the failover secret. If you use an ARN, the Region in the ARN must match the field `failoverRegion`.

objectVersion

(Optional) The version ID of the secret. Must match the primary `objectVersion`. Not recommended because you must update the version ID every time you update the secret. By default the most recent version is used.

objectVersionLabel

(Optional) The alias for the version. The default is the most recent version AWSCURRENT. For more information, see the section called “Secret versions”.

Create a basic SecretProviderClass configuration to mount secrets in your Amazon EKS Pods.

Pod Identity

SecretProviderClass to use a secret in the same Amazon EKS cluster:

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: aws-secrets-manager
spec:
  provider: aws
  parameters:
    objects: |
      - objectName: "mySecret"
        objectType: "secretsmanager"
    usePodIdentity: "true"
```

IRSA

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: deployment-aws-secrets
spec:
  provider: aws
  parameters:
    objects: |
        - objectName: "MySecret"
          objectType: "secretsmanager"
```

SecretProviderClass usage

Use these examples to create SecretProviderClass configurations for different scenarios.

Example: Mount secrets by name or ARN

This example shows how to mount three different types of secrets:

- A secret specified by full ARN
- A secret specified by name
- A specific version of a secret

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: aws-secrets
spec:
  provider: aws
  parameters:
    objects: |
      - objectName: "arn:aws:secretsmanager:us-east-2:777788889999:secret:MySecret2-
d4e5f6"
      - objectName: "MySecret3"
        objectType: "secretsmanager"
      - objectName: "MySecret4"
        objectType: "secretsmanager"
        objectVersionLabel: "AWSCURRENT"
```

Example: Mount key-value pairs from a secret

This example shows how to mount specific key-value pairs from a JSON-formatted secret:

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: aws-secrets
spec:
  provider: aws
  parameters:
    objects: |
      - objectName: "arn:aws:secretsmanager:us-east-2:777788889999:secret:MySecret-
a1b2c3"
        jmesPath:
            - path: username
              objectAlias: dbusername
            - path: password
              objectAlias: dbpassword
```

Example: Mount secrets by file permission

This example shows how to mount a secret with a specific file permission

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
```

```
metadata:
  name: aws-secrets
spec:
  provider: aws
  parameters:
    objects: |
      - objectName: "mySecret"
        objectType: "secretsmanager"
        filePermission: "0600"
        jmesPath:
            - path: username
              objectAlias: dbusername
              filePermission: "0400"
```

Example: Failover configuration examples

These examples show how to configure failover for secrets.

Multi-Region secret failover

This example shows how to configure automatic failover for a secret replicated across multiple Regions:

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: aws-secrets
spec:
  provider: aws
  parameters:
    region: us-east-1
    failoverRegion: us-east-2
    objects: |
      - objectName: "MySecret"
```

Failover to a different secret

This example shows how to configure failover to a different secret (not a replica):

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: aws-secrets
```

```
spec:
  provider: aws
  parameters:
    region: us-east-1
    failoverRegion: us-east-2
    objects: |
      - objectName: "arn:aws:secretsmanager:us-east-1:777788889999:secret:MySecret-
a1b2c3"
        objectAlias: "MyMountedSecret"
        failoverObject:
          - objectName: "arn:aws:secretsmanager:us-
east-2:777788889999:secret:MyFailoverSecret-d4e5f6"
```

Additional resources

For more information about using ASCP with Amazon EKS, see the following resources:

- Using Pod Identity with Amazon EKS
- Using AWS Secrets and Configuration Provider
- AWS Secrets Store CSI Driver on GitHub

Use AWS Secrets Manager secrets in AWS Lambda functions

AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers. Parameter Store, a capability of AWS Systems Manager, provides secure, hierarchical storage for configuration data management and secrets management. You can use the AWS Parameters and Secrets Lambda Extension to retrieve and cache AWS Secrets Manager secrets and Parameter Store parameters in Lambda functions without using an SDK. For detailed information about using this extension, see Use Secrets Manager secrets in Lambda functions in the *Lambda Developer Guide*.

Using Secrets Manager secrets with Lambda

The Lambda Developer Guide provides comprehensive instructions for using Secrets Manager secrets in Lambda functions. To get started:

1. Follow the step-by-step tutorial in Use Secrets Manager secrets in Lambda functions, which includes:
   - Creating a Lambda function with your preferred runtime (Python, Node.js, Java)

  - Adding the AWS Parameters and Secrets Lambda Extension as a layer
  - Configuring the necessary permissions
  - Writing code to retrieve secrets from the extension
  - Testing your function
2. Learn about environment variables for configuring the extension's behavior, including cache settings and timeouts
3. Understand best practices for working with secret rotation

Using Secrets Manager and Lambda in a VPC

If your Lambda function runs in a VPC, you need to create a VPC endpoint so that the extension can make calls to Secrets Manager. For more information, see the section called "VPC endpoints (AWS PrivateLink)".

Using the AWS Parameters and Secrets Lambda Extension

The extension can retrieve both Secrets Manager secrets and Parameter Store parameters. For detailed information about using Parameter Store parameters with the extension, see Using Parameter Store parameters in Lambda functions in the *AWS Systems Manager User Guide*.

The Systems Manager documentation includes:

- Detailed explanation of how the extension works with Parameter Store
- Instructions for adding the extension to a Lambda function
- Environment variables for configuring the extension
- Sample commands for retrieving parameters
- Complete list of extension ARNs for all supported architectures and regions

Using the AWS Secrets Manager Agent

How the Secrets Manager Agent works

The AWS Secrets Manager Agent is a client-side HTTP service that helps you standardize how you consume secrets from Secrets Manager across your compute environments. You can use it with the following services:

- AWS Lambda
- Amazon Elastic Container Service
- Amazon Elastic Kubernetes Service
- Amazon Elastic Compute Cloud

The Secrets Manager Agent retrieves and caches secrets in memory, allowing your applications to get secrets from localhost instead of making direct calls to Secrets Manager. The Secrets Manager Agent can only read secrets—it can't modify them.

> **Important**
>
> The Secrets Manager Agent uses the AWS credentials from your environment to call Secrets Manager. It includes protection against Server Side Request Forgery (SSRF) to help improve secret security. The Secrets Manager Agent uses the post-quantum ML-KEM key exchange as the highest-priority key exchange by default.

Understanding Secrets Manager Agent caching

The Secrets Manager Agent uses an in-memory cache that resets when the Secrets Manager Agent restarts. It periodically refreshes cached secret values based on the following:

- The default refresh frequency (TTL) is 300 seconds
- You can modify the TTL using a configuration file
- The refresh occurs when you request a secret after the TTL expires

> **Note**
>
> The Secrets Manager Agent doesn't include cache invalidation. If a secret rotates before the cache entry expires, the Secrets Manager Agent might return a stale secret value.

The Secrets Manager Agent returns secret values in the same format as the response of `GetSecretValue`. Secret values aren't encrypted in the cache.

Topics

Build the Secrets Manager Agent

Before you begin, ensure you have the standard development tools and Rust tools installed for your platform.

> **Note**
>
> Building the agent with the `fips` feature enabled on macOS currently requires the following workaround:
>
> - Create an environment variable called `SDKROOT` which is set to the result of running `xcrun --show-sdk-path`

RPM-based systems

To build on RPM-based systems

1. Use the `install` script provided in the repository.

   The script generates a random SSRF token on startup and stores it in the file `/var/run/awssmatoken`. The token is readable by the `awssmatokenreader` group that the install script creates.

2. To allow your application to read the token file, you need to add the user account that your application runs under to the `awssmatokenreader` group. For example, you can grant permissions for your application to read the token file with the following usermod command, where *<APP_USER>* is the user ID under which your application runs.

```
sudo usermod -aG awssmatokenreader <APP_USER>
```

Install development tools

On RPM-based systems such as AL2023, install the Development Tools group:

```
sudo yum -y groupinstall "Development Tools"
```

3. **Install Rust**

   Follow the instructions at [Install Rust](#) in the *Rust documentation*:

   ```
   curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh # Follow the on-
   screen instructions
   . "$HOME/.cargo/env"
   ```

4. **Build the agent**

   Build the Secrets Manager Agent using the cargo build command:

   ```
   cargo build --release
   ```

   You will find the executable under `target/release/aws_secretsmanager_agent`.

Debian-based systems

To build on Debian-based systems

1. **Install development tools**

   On Debian-based systems such as Ubuntu, install the build-essential package:

   ```
   sudo apt install build-essential
   ```

2. **Install Rust**

   Follow the instructions at [Install Rust](#) in the *Rust documentation*:

   ```
   curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh # Follow the on-
   screen instructions
   ```

```
. "$HOME/.cargo/env"
```

3. **Build the agent**

   Build the Secrets Manager Agent using the cargo build command:

   ```
   cargo build --release
   ```

   You will find the executable under `target/release/aws_secretsmanager_agent`.

Windows

To build on Windows

1. **Set up development environment**

   Follow the instructions at [Set up your dev environment on Windows for Rust] in the *Microsoft Windows documentation*.

2. **Build the agent**

   Build the Secrets Manager Agent using the cargo build command:

   ```
   cargo build --release
   ```

   You will find the executable under `target/release/aws_secretsmanager_agent.exe`.

Cross-compile natively

To cross-compile natively

1. **Install cross-compile tools**

   On distributions where the mingw-w64 package is available such as Ubuntu, install the cross-compile toolchain:

   ```
   # Install the cross compile tool chain
   sudo add-apt-repository universe
   sudo apt install -y mingw-w64
   ```

2. **Add Rust build targets**

   Install the Windows GNU build target:

   ```
   rustup target add x86_64-pc-windows-gnu
   ```

3. **Build for Windows**

   Cross-compile the agent for Windows:

   ```
   cargo build --release --target x86_64-pc-windows-gnu
   ```

   You will find the executable at `target/x86_64-pc-windows-gnu/release/aws_secretsmanager_agent.exe`.

Cross compile with Rust cross

To cross-compile using Rust cross

If the cross-compile tools are not available natively on the system, you can use the Rust cross project. For more information, see https://github.com/cross-rs/cross.

> **Important**
>
> We recommend 32GB disk space for the build environment.

1. **Set up Docker**

   Install and configure Docker:

   ```
   # Install and start docker
   sudo yum -y install docker
   sudo systemctl start docker
   sudo systemctl enable docker # Make docker start after reboot
   ```

2. **Configure Docker permissions**

   Add your user to the docker group:

   ```
   # Give ourselves permission to run the docker images without sudo
   ```

```
sudo usermod -aG docker $USER
newgrp docker
```

3. **Build for Windows**

   Install cross and build the executable:

   ```
   # Install cross and cross compile the executable
   cargo install cross
   cross build --release --target x86_64-pc-windows-gnu
   ```

Install the Secrets Manager Agent

Choose your compute environment from the following installation options.

Amazon EC2

To install the Secrets Manager Agent on Amazon EC2

1. **Navigate to configuration directory**

   Change to the configuration directory:

   ```
   cd aws_secretsmanager_agent/configuration
   ```

2. **Run installation script**

   Run the `install` script provided in the repository.

   The script generates a random SSRF token on startup and stores it in the file `/var/run/awssmatoken`. The token is readable by the `awssmatokenreader` group that the install script creates.

3. **Configure application permissions**

   Add the user account that your application runs under to the `awssmatokenreader` group:

   ```
   sudo usermod -aG awssmatokenreader APP_USER
   ```

   Replace *APP_USER* with the user ID under which your application runs.

Container Sidecar

You can run the Secrets Manager Agent as a sidecar container alongside your application by using Docker. Then your application can retrieve secrets from the local HTTP server the Secrets Manager Agent provides. For information about Docker, see the Docker documentation.

To create a sidecar container for the Secrets Manager Agent

1. **Create agent Dockerfile**

   Create a Dockerfile for the Secrets Manager Agent sidecar container:

   ```
   # Use the latest Debian image as the base
   FROM debian:latest

   # Set the working directory inside the container
   WORKDIR /app

   # Copy the Secrets Manager Agent binary to the container
   COPY secrets-manager-agent .

   # Install any necessary dependencies
   RUN apt-get update && apt-get install -y ca-certificates

   # Set the entry point to run the Secrets Manager Agent binary
   ENTRYPOINT ["./secrets-manager-agent"]
   ```

2. **Create application Dockerfile**

   Create a Dockerfile for your client application.

3. **Create Docker Compose file**

   Create a Docker Compose file to run both containers with a shared network interface:

   > **Important**
   >
   > You must load AWS credentials and the SSRF token for the application to be able to use the Secrets Manager Agent. For Amazon EKS and Amazon ECS, see the following:
   >
   > - Manage access in the *Amazon EKS User Guide*

- [Amazon ECS task IAM role] in the *Amazon ECS Developer Guide*

```
version: '3'
services:
    client-application:
    container_name: client-application
    build:
        context: .
        dockerfile: Dockerfile.client
    command: tail -f /dev/null  # Keep the container running


    secrets-manager-agent:
    container_name: secrets-manager-agent
    build:
        context: .
        dockerfile: Dockerfile.agent
    network_mode: "container:client-application"  # Attach to the client-
application container's network
    depends_on:
        - client-application
```

4. **Copy agent binary**

   Copy the `secrets-manager-agent` binary to the same directory that contains your Dockerfiles and Docker Compose file.

5. **Build and run containers**

   Build and run the containers using Docker Compose:

   ```
   docker-compose up --build
   ```

6. **Next steps**

   You can now use the Secrets Manager Agent to retrieve secrets from your client container. For more information, see the section called "Retrieve secrets with the Secrets Manager Agent".

Lambda

You can package the Secrets Manager Agent as a Lambda extension. Then you can add it to your Lambda function as a layer and call the Secrets Manager Agent from your Lambda function to get secrets.

The following instructions show how to get a secret named *MyTest* by using the example script `secrets-manager-agent-extension.sh` in https://github.com/aws/aws-secretsmanager-agent to install the Secrets Manager Agent as a Lambda extension.

To create a Lambda extension for the Secrets Manager Agent

1. **Package the agent layer**

   From the root of the Secrets Manager Agent code package, run the following commands:

```
AWS_ACCOUNT_ID=AWS_ACCOUNT_ID
LAMBDA_ARN=LAMBDA_ARN

# Build the release binary
cargo build --release --target=x86_64-unknown-linux-gnu

# Copy the release binary into the `bin` folder
mkdir -p ./bin
cp ./target/x86_64-unknown-linux-gnu/release/aws_secretsmanager_agent ./bin/
secrets-manager-agent

# Copy the `secrets-manager-agent-extension.sh` example script into the
 `extensions` folder.
mkdir -p ./extensions
cp aws_secretsmanager_agent/examples/example-lambda-extension/secrets-manager-
agent-extension.sh ./extensions

# Zip the extension shell script and the binary
zip secrets-manager-agent-extension.zip bin/* extensions/*

# Publish the layer version
LAYER_VERSION_ARN=$(aws lambda publish-layer-version \
    --layer-name secrets-manager-agent-extension \
    --zip-file "fileb://secrets-manager-agent-extension.zip" | jq -r
 '.LayerVersionArn')
```

2. **Configure SSRF token**

The default configuration of the agent will automatically set the SSRF token to the value set in the pre-set `AWS_SESSION_TOKEN` or `AWS_CONTAINER_AUTHORIZATION_TOKEN` environment variables (the latter variable for Lambda functions with SnapStart enabled). Alternatively, you can define the `AWS_TOKEN` environment variable with an arbitrary value for your Lambda function instead as this variable takes precedence over the other two. If you choose to use the `AWS_TOKEN` environment variable, you must set that environment variable with a `lambda:UpdateFunctionConfiguration` call.

3. **Attach layer to function**

   Attach the layer version to your Lambda function:

```
# Attach the layer version to the Lambda function
aws lambda update-function-configuration \
    --function-name $LAMBDA_ARN \
    --layers "$LAYER_VERSION_ARN"
```

4. **Update function code**

   Update your Lambda function to query `http://localhost:2773/secretsmanager/get?secretId=MyTest` with the `X-Aws-codes-Secrets-Token` header value set to the value of the SSRF token sourced from one the environment variables mentioned above to retrieve the secret. Be sure to implement retry logic in your application code to accommodate delays in initialization and registration of the Lambda extension.

5. **Test the function**

   Invoke the Lambda function to verify that the secret is being correctly fetched.

Retrieve secrets with the Secrets Manager Agent

To retrieve a secret, call the local Secrets Manager Agent endpoint with the secret name or ARN as a query parameter. By default, the Secrets Manager Agent retrieves the `AWSCURRENT` version of the secret. To retrieve a different version, use either the versionStage or versionId parameter.

> **Important**
>
> To help protect the Secrets Manager Agent, you must include a SSRF token header as part of each request: `X-Aws-Parameters-Secrets-Token`. The Secrets Manager Agent

denies requests that don't have this header or that have an invalid SSRF token. You can customize the SSRF header name in the the section called “Configuration options”.

Required permissions

The Secrets Manager Agent uses the AWS SDK for Rust, which uses the AWS credential provider chain. The identity of these IAM credentials determines the permissions the Secrets Manager Agent has to retrieve secrets.

- `secretsmanager:DescribeSecret`
- `secretsmanager:GetSecretValue`

For more information about permissions, see the section called “Permissions reference”.

> **Important**
>
> After the secret value is pulled into the Secrets Manager Agent, any user with access to the compute environment and SSRF token can access the secret from the Secrets Manager Agent cache. For more information, see the section called “Security considerations”.

Example requests

curl

Example – Get a secret using curl

The following curl example shows how to get a secret from the Secrets Manager Agent. The example relies on the SSRF being present in a file, which is where it is stored by the install script.

```
curl -v -H \\
    "X-Aws-Parameters-Secrets-Token: $(/var/run/awssmatoken)" \\
    'http://localhost:2773/secretsmanager/get?secretId=YOUR_SECRET_ID' \\
    echo
```

Python

Example – Get a secret using Python

The following Python example shows how to get a secret from the Secrets Manager Agent. The example relies on the SSRF being present in a file, which is where it is stored by the install script.

```
import requests
import json

# Function that fetches the secret from Secrets Manager Agent for the provided
 secret id.
def get_secret():
    # Construct the URL for the GET request
    url = f"http://localhost:2773/secretsmanager/get?secretId=YOUR_SECRET_ID"

    # Get the SSRF token from the token file
    with open('/var/run/awssmatoken') as fp:
        token = fp.read()

    headers = {
        "X-Aws-Parameters-Secrets-Token": token.strip()
    }

    try:
        # Send the GET request with headers
        response = requests.get(url, headers=headers)

        # Check if the request was successful
        if response.status_code == 200:
            # Return the secret value
            return response.text
        else:
            # Handle error cases
            raise Exception(f"Status code {response.status_code} - {response.text}")

    except Exception as e:
        # Handle network errors
        raise Exception(f"Error: {e}")
```

Understanding the `refreshNow` parameter

The Secrets Manager Agent uses an in-memory cache to store secret values, which it refreshes periodically. By default, this refresh occurs when you request a secret after the Time to Live (TTL) has expired, typically every 300 seconds. However, this approach can sometimes result in stale secret values, especially if a secret rotates before the cache entry expires.

To address this limitation, the Secrets Manager Agent supports a parameter called `refreshNow` in the URL. You can use this parameter to force an immediate refresh of a secret's value, bypassing the cache and ensuring you have the most up-to-date information.

Default behavior (without `refreshNow`)

- Uses cached values until TTL expires
- Refreshes secrets only after TTL (default 300 seconds)
- May return stale values if secrets rotate before the cache expires

Behavior with `refreshNow=true`

- Bypasses the cache entirely
- Retrieves the latest secret value directly from Secrets Manager
- Updates the cache with the fresh value and resets the TTL
- Ensures you always get the most current secret value

Force-refresh a secret value

Important

The default value of `refreshNow` is `false`. When set to `true`, it overrides the TTL specified in the Secrets Manager Agent configuration file and makes an API call to Secrets Manager.

curl

Example – Force-refresh a secret using curl

The following curl example shows how to force the Secrets Manager Agent to refresh the secret. The example relies on the SSRF being present in a file, which is where it is stored by the install script.

```
curl -v -H \\
"X-Aws-Parameters-Secrets-Token: $(/var/run/awssmatoken)" \\
'http://localhost:2773/secretsmanager/get?secretId=YOUR_SECRET_ID&refreshNow=true' \
\
echo
```

Python

Example – Force-refresh a secret using Python

The following Python example shows how to get a secret from the Secrets Manager Agent. The example relies on the SSRF being present in a file, which is where it is stored by the install script.

```
import requests
import json

# Function that fetches the secret from Secrets Manager Agent for the provided
 secret id.
def get_secret():
    # Construct the URL for the GET request
    url = f"http://localhost:2773/secretsmanager/get?
secretId=YOUR_SECRET_ID&refreshNow=true"

    # Get the SSRF token from the token file
    with open('/var/run/awssmatoken') as fp:
        token = fp.read()

    headers = {
        "X-Aws-Parameters-Secrets-Token": token.strip()
    }

    try:
        # Send the GET request with headers
        response = requests.get(url, headers=headers)

        # Check if the request was successful
        if response.status_code == 200:
            # Return the secret value
            return response.text
        else:
            # Handle error cases
            raise Exception(f"Status code {response.status_code} - {response.text}")
```

```
    except Exception as e:
        # Handle network errors
        raise Exception(f"Error: {e}")
```

Configure the Secrets Manager Agent

To change the configuration of the Secrets Manager Agent, create a [TOML](https://toml.io/en/) config file, and then call `./aws_secretsmanager_agent --config config.toml`.

Configuration options

`log_level`

The level of detail reported in logs for the Secrets Manager Agent: DEBUG, INFO, WARN, ERROR, or NONE. The default is INFO.

`log_to_file`

Whether to log to a file or stdout/stderr: `true` or `false`. The default is `true`.

`http_port`

The port for the local HTTP server, in the range 1024 to 65535. The default is 2773.

`region`

The AWS Region to use for requests. If no Region is specified, the Secrets Manager Agent determines the Region from the SDK. For more information, see Specify your credentials and default Region in the *AWS SDK for Rust Developer Guide*.

`ttl_seconds`

The TTL in seconds for the cached items, in the range 0 to 3600. The default is 300. 0 indicates that there is no caching.

`cache_size`

The maximum number of secrets that can be stored in the cache, in the range 1 to 1000. The default is 1000.

`ssrf_headers`

A list of header names the Secrets Manager Agent checks for the SSRF token. The default is "X-Aws-Parameters-Secrets-Token, X-Vault-Token".

`ssrf_env_variables`

A list of environment variable names the Secrets Manager Agent checks in sequential order for the SSRF token. The environment variable can contain the token or a reference to the token file as in: `AWS_TOKEN=file:///var/run/awssmatoken`. The default is "AWS_TOKEN, AWS_SESSION_TOKEN, AWS_CONTAINER_AUTHORIZATION_TOKEN".

`path_prefix`

The URI prefix used to determine if the request is a path based request. The default is "/v1/".

`max_conn`

The maximum number of connections from HTTP clients that the Secrets Manager Agent allows, in the range 1 to 1000. The default is 800.

Optional features

The Secrets Manager Agent can be built with optional features by passing the `--features` flag to `cargo build`. The available features are:

Build features

`prefer-post-quantum`

Makes `X25519MLKEM768` the highest-priority key exchange algorithm. Otherwise, it is available but not highest-priority. `X25519MLKEM768` is a hybrid, post-quantum-secure key exchange algorithm.

`fips`

Restricts the cipher suites used by the agent to only FIPS-approved ciphers.

Logging

Local logging

The Secrets Manager Agent logs errors locally to the file `logs/secrets_manager_agent.log` or to stdout/stderr depending on the `log_to_file` config variable. When your application calls the Secrets Manager Agent to get a secret, those calls appear in the local log. They do not appear in the CloudTrail logs.

Log rotation

The Secrets Manager Agent creates a new log file when the file reaches 10 MB, and it stores up to five log files total.

AWS service logging

The log does not go to Secrets Manager, CloudTrail, or CloudWatch. Requests to get secrets from the Secrets Manager Agent do not appear in those logs. When the Secrets Manager Agent makes a call to Secrets Manager to get a secret, that call is recorded in CloudTrail with a user agent string containing `aws-secrets-manager-agent`.

You can configure logging options in the the section called "Configuration options".

Security considerations

Domain of trust

For an agent architecture, the domain of trust is where the agent endpoint and SSRF token are accessible, which is usually the entire host. The domain of trust for the Secrets Manager Agent should match the domain where the Secrets Manager credentials are available in order to maintain the same security posture. For example, on Amazon EC2 the domain of trust for the Secrets Manager Agent would be the same as the domain of the credentials when using roles for Amazon EC2.

> **Important**
>
> Security conscious applications that are not already using an agent solution with the Secrets Manager credentials locked down to the application should consider using the language-specific AWS SDKs or caching solutions. For more information, see Get secrets.

Get a Secrets Manager secret value using the C++ AWS SDK

For C++ applications, call the SDK directly with GetSecretValue or BatchGetSecretValue.

The following code example shows how to get a Secrets Manager secret value.

Required permissions: `secretsmanager:GetSecretValue`

```
//! Retrieve an AWS Secrets Manager encrypted secret.
/*!
  \param secretID: The ID for the secret.
  \return bool: Function succeeded.
 */
bool AwsDoc::SecretsManager::getSecretValue(const Aws::String &secretID,
                                            const Aws::Client::ClientConfiguration
 &clientConfiguration) {
    Aws::SecretsManager::SecretsManagerClient
 secretsManagerClient(clientConfiguration);

    Aws::SecretsManager::Model::GetSecretValueRequest request;
    request.SetSecretId(secretID);

    Aws::SecretsManager::Model::GetSecretValueOutcome getSecretValueOutcome =
 secretsManagerClient.GetSecretValue(
            request);
    if (getSecretValueOutcome.IsSuccess()) {
        std::cout << "Secret is: "
                  << getSecretValueOutcome.GetResult().GetSecretString() << std::endl;
    }
    else {
        std::cerr << "Failed with Error: " << getSecretValueOutcome.GetError()
                  << std::endl;
    }

    return getSecretValueOutcome.IsSuccess();
}
```

Get a Secrets Manager secret value using the JavaScript AWS SDK

For JavaScript applications, call the SDK directly with `getSecretValue` or `batchGetSecretValue`.

The following code example shows how to get a Secrets Manager secret value.

Required permissions: `secretsmanager:GetSecretValue`

```
import {
  GetSecretValueCommand,
```

```
  SecretsManagerClient,
} from "@aws-sdk/client-secrets-manager";

export const getSecretValue = async (secretName = "SECRET_NAME") => {
  const client = new SecretsManagerClient();
  const response = await client.send(
    new GetSecretValueCommand({
      SecretId: secretName,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '584eb612-f8b0-48c9-855e-6d246461b604',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   ARN: 'arn:aws:secretsmanager:us-east-1:xxxxxxxxxxxx:secret:binary-
secret-3873048-xxxxxx',
  //   CreatedDate: 2023-08-08T19:29:51.294Z,
  //   Name: 'binary-secret-3873048',
  //   SecretBinary: Uint8Array(11) [
  //      98, 105, 110, 97, 114,
  //     121,  32, 100, 97, 116,
  //      97
  //   ],
  //   VersionId: '712083f4-0d26-415e-8044-16735142cd6a',
  //   VersionStages: [ 'AWSCURRENT' ]
  // }

  if (response.SecretString) {
    return response.SecretString;
  }

  if (response.SecretBinary) {
    return response.SecretBinary;
  }
};
```

Get a Secrets Manager secret value using the Kotlin AWS SDK

For Kotlin applications, call the SDK directly with [GetSecretValue] or [BatchGetSecretValue].

The following code example shows how to get a Secrets Manager secret value.

Required permissions: `secretsmanager:GetSecretValue`

```
suspend fun getValue(secretName: String?) {
    val valueRequest =
        GetSecretValueRequest {
            secretId = secretName
        }

    SecretsManagerClient.fromEnvironment { region = "us-east-1" }.use { secretsClient ->
        val response = secretsClient.getSecretValue(valueRequest)
        val secret = response.secretString
        println("The secret value is $secret")
    }
}
```

Get a Secrets Manager secret value using the PHP AWS SDK

For PHP applications, call the SDK directly with `GetSecretValue` or `BatchGetSecretValue`.

The following code example shows how to get a Secrets Manager secret value.

Required permissions: `secretsmanager:GetSecretValue`

```
<?php

  /**
    * Use this code snippet in your app.
    *
    * If you need more information about configurations or implementing the sample
 code, visit the AWS docs:
    * https://aws.amazon.com/developer/language/php/
    */

  require 'vendor/autoload.php';
```

```
use Aws\SecretsManager\SecretsManagerClient;
use Aws\Exception\AwsException;

/**
  * This code expects that you have AWS credentials set up per:
  * https://<<{{DocsDomain}}>>/sdk-for-php/v3/developer-guide/guide_credentials.html
  */

// Create a Secrets Manager Client
$client = new SecretsManagerClient([
    'profile' => 'default',
    'version' => '2017-10-17',
    'region' => '<<{{MyRegionName}}>>',
]);

$secret_name = '<<{{MySecretName}}>>';

try {
    $result = $client->getSecretValue([
        'SecretId' => $secret_name,
    ]);
} catch (AwsException $e) {
    // For a list of exceptions thrown, see
    // https://<<{{DocsDomain}}>>/secretsmanager/latest/apireference/
API_GetSecretValue.html
    throw $e;
}

// Decrypts secret using the associated KMS key.
$secret = $result['SecretString'];

// Your code goes here
```

Get a Secrets Manager secret value using the Ruby AWS SDK

For Ruby applications, call the SDK directly with `get_secret_value` or `batch_get_secret_value`.

The following code example shows how to get a Secrets Manager secret value.

Required permissions: `secretsmanager:GetSecretValue`

```
# Use this code snippet in your app.
```

```
  # If you need more information about configurations or implementing the sample code,
 visit the AWS docs:
  # https://aws.amazon.com/developer/language/ruby/

  require 'aws-sdk-secretsmanager'

  def get_secret
    client = Aws::SecretsManager::Client.new(region: '<<{{MyRegionName}}>>')

    begin
      get_secret_value_response = client.get_secret_value(secret_id:
 '<<{{MySecretName}}>>')
    rescue StandardError => e
      # For a list of exceptions thrown, see
      # https://<<{{DocsDomain}}>>/secretsmanager/latest/apireference/
API_GetSecretValue.html
      raise e
    end

    secret = get_secret_value_response.secret_string
    # Your code goes here.
  end
```

Get a secret value using the AWS CLI

Required permissions: `secretsmanager:GetSecretValue`

Example Retrieve the encrypted secret value of a secret

The following get-secret-value example gets the current secret value.

```
aws secretsmanager get-secret-value \
    --secret-id MyTestSecret
```

Example Retrieve the previous secret value

The following get-secret-value example gets the previous secret value.

```
aws secretsmanager get-secret-value \
        --secret-id MyTestSecret
        --version-stage AWSPREVIOUS
```

Get a group of secrets in a batch using the AWS CLI

Required permissions:

- `secretsmanager:BatchGetSecretValue`
- `secretsmanager:GetSecretValue` permission for each secret you want to retrieve.
- If you use filters, you must also have `secretsmanager:ListSecrets`.

For an example permissions policy, see the section called "Example: Permission to retrieve a group of secret values in a batch".

> **Important**
>
> If you have a VPCE policy that denies permission to retrieve an individual secret in the group you are retrieving, `BatchGetSecretValue` will not return any secret values, and it will return an error.

Example Retrieve the secret value for a group of secrets listed by name

The following `batch-get-secret-value` example gets the secret value for three secrets.

```
aws secretsmanager batch-get-secret-value \
          --secret-id-list MySecret1 MySecret2 MySecret3
```

Example Retrieve the secret value for a group of secrets selected by filter

The following `batch-get-secret-value` example gets the secret value for the secrets that have a tag named "Test".

```
aws secretsmanager batch-get-secret-value \
          --filters Key="tag-key",Values="Test"
```

Get a secret value using the AWS console

To retrieve a secret (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.

2. In the list of secrets, choose the secret you want to retrieve.
3. In the **Secret value** section, choose **Retrieve secret value**.

   Secrets Manager displays the current version (`AWSCURRENT`) of the secret. To see other versions of the secret, such as `AWSPREVIOUS` or custom labeled versions, use the the section called "AWS CLI".

Use AWS Secrets Manager secrets in AWS Batch

AWS Batch helps you to run batch computing workloads on the AWS Cloud. With AWS Batch, you can inject sensitive data into your jobs by storing your sensitive data in AWS Secrets Manager secrets and then referencing them in your job definition. For more information, see Specifying sensitive data using Secrets Manager.

Get an AWS Secrets Manager secret in an CloudFormation resource

With CloudFormation, you can retrieve a secret to use in another CloudFormation resource. A common scenario is to first create a secret with a password generated by Secrets Manager, and then retrieve the username and password from the secret to use as credentials for a new database. For information about creating secrets with CloudFormation, see *CloudFormation*.

To retrieve a secret in an CloudFormation template, you use a *dynamic reference*. When you create the stack, the dynamic reference pulls the secret value into the CloudFormation resource, so you don't have to hardcode the secret information. Instead, you refer to the secret by name or ARN. You can use a dynamic reference for a secret in any resource property. You can't use a dynamic reference for a secret in resource metadata such as `AWS::CloudFormation::Init` because that would make the secret value visible in the console.

A dynamic reference for a secret has the following pattern:

```
{{resolve:secretsmanager:secret-id:SecretString:json-key:version-stage:version-id}}
```

secret-id

The name or ARN of the secret. To access a secret in your AWS account, you can use the secret name. To access a secret in a different AWS account, use the ARN of the secret.

json-key (Optional)

The key name of the key-value pair whose value you want to retrieve. If you don't specify a `json-key`, CloudFormation retrieves the entire secret text. This segment may not include the colon character (`:`).

version-stage (Optional)

The version of the secret to use. Secrets Manager uses staging labels to keep track of different versions during the rotation process. If you use `version-stage` then don't specify `version-id`. If you don't specify either `version-stage` or `version-id`, then the default is the `AWSCURRENT` version. This segment may not include the colon character (`:`).

version-id (Optional)

The unique identifier of the version of the secret to use. If you specify `version-id`, then don't specify `version-stage`. If you don't specify either `version-stage` or `version-id`, then the default is the `AWSCURRENT` version. This segment may not include the colon character (`:`).

For more information, see Using dynamic references to specify Secrets Manager secrets.

Note

Do not create a dynamic reference using a backslash `(\)` as the final value. CloudFormation can't resolve those references, which causes a resource failure.

Use AWS Secrets Manager secrets in GitHub jobs

To use a secret in a GitHub job, you can use a GitHub action to retrieve secrets from AWS Secrets Manager and add them as masked Environment variables in your GitHub workflow. For more information about GitHub Actions, see Understanding GitHub Actions in the *GitHub Docs*.

When you add a secret to your GitHub environment, it is available to all other steps in your GitHub job. Follow the guidance in Security hardening for GitHub Actions to help prevent secrets in your environment from being misused.

You can set the entire string in the secret value as the environment variable value, or if the string is JSON, you can parse the JSON to set individual environment variables for each JSON key-value pair. If the secret value is a binary, the action converts it to a string.

To view the environment variables created from your secrets, turn on debug logging. For more information, see Enabling debug logging in the *GitHub Docs*.

To use the environment variables created from your secrets, see Environment variables in the *GitHub Docs*.

Prerequisites

To use this action, you first need to configure AWS credentials and set the AWS Region in your GitHub environment by using the `configure-aws-credentials` step. Follow the instructions in Configure AWS Credentials Action For GitHub Actions to **Assume role directly using GitHub OIDC provider**. This allows you to use short-lived credentials and avoid storing additional access keys outside of Secrets Manager.

The IAM role the action assumes must have the following permissions:

- `GetSecretValue` on the secrets you want to retrieve.
- `ListSecrets` on all secrets.
- (Optional) `Decrypt` on the KMS key if the secrets are encrypted with a customer managed key.

For more information, see the section called "Authentication and access control".

Usage

To use the action, add a step to your workflow that uses the following syntax.

```
- name: Step name
  uses: aws-actions/aws-secretsmanager-get-secrets@v2
  with:
    secret-ids: |
      secretId1
      ENV_VAR_NAME, secretId2
    name-transformation: (Optional) uppercase|lowercase|none
    parse-json-secrets: (Optional) true|false
```

Parameters

`secret-ids`

Secret ARNS, names, and name prefixes.

To set the environment variable name, enter it before the secret ID, followed by a comma. For example `ENV_VAR_1, secretId` creates an environment variable named **ENV_VAR_1** from the secret `secretId`. The environment variable name can consist of uppercase letters, numbers, and underscores.

To use a prefix, enter at least three characters followed by an asterisk. For example `dev*` matches all secrets with a name beginning in **dev**. The maximum number of matching secrets that can be retrieved is 100. If you set the variable name, and the prefix matches multiple secrets, then the action fails.

`name-transformation`

By default, the step creates each environment variable name from the secret name, transformed to include only uppercase letters, numbers, and underscores, and so that it doesn't begin with a number. For the letters in the name, you can configure the step to use lowercase letters with `lowercase` or to not change the case of the letters with `none`. The default value is `uppercase`.

`parse-json-secrets`

(Optional) By default, the action sets the environment variable value to the entire JSON string in the secret value. Set `parse-json-secrets` to `true` to create environment variables for each key-value pair in the JSON.

Note that if the JSON uses case-sensitive keys such as "name" and "Name", the action will have duplicate name conflicts. In this case, set `parse-json-secrets` to `false` and parse the JSON secret value separately.

Environment variable naming

The environment variables created by the action are named the same as the secrets that they come from. Environment variables have stricter naming requirements than secrets, so the action transforms secret names to meet those requirements. For example, the action transforms lowercase letters to uppercase letters. If you parse the JSON of the secret, then the environment variable name includes both the secret name and the JSON key name, for example `MYSECRET_KEYNAME`. You can configure the action to not transform lowercase letters.

If two environment variables would end up with the same name, the action fails. In this case, you must specify the names you want to use for the environment variables as *aliases*.

Examples of when the names might conflict:

- A secret named "MySecret" and a secret named "mysecret" would both become environment variables named "MYSECRET".
- A secret named "Secret_keyname" and a JSON-parsed secret named "Secret" with a key named "keyname" would both become environment variables named "SECRET_KEYNAME".

You can set the environment variable name by specifying an *alias*, as shown in the following example, which creates a variable named `ENV_VAR_NAME`.

```
secret-ids: |
  ENV_VAR_NAME, secretId2
```

Blank aliases

- If you set `parse-json-secrets: true` and enter a blank alias, followed by a comma and then the secret ID, the action names the environment variable the same as the parsed JSON keys. The variable names do not include the secret name.

  If the secret doesn't contain valid JSON, then the action creates one environment variable and names it the same as the secret name.
- If you set `parse-json-secrets: false` and enter a blank alias, followed by a comma and the secret ID, the action names the environment variables as if you did not specify an alias.

The following example shows a blank alias.

```
,secret2
```

Examples

Example 1 Get secrets by name and by ARN

The following example creates environment variables for secrets identified by name and by ARN.

```
- name: Get secrets by name and by ARN
  uses: aws-actions/aws-secretsmanager-get-secrets@v2
  with:
    secret-ids: |
```

```
      exampleSecretName
      arn:aws:secretsmanager:us-east-2:123456789012:secret:test1-a1b2c3
      0/test/secret
      /prod/example/secret
      SECRET_ALIAS_1,test/secret
      SECRET_ALIAS_2,arn:aws:secretsmanager:us-east-2:123456789012:secret:test2-a1b2c3
      ,secret2
```

Environment variables created:

```
EXAMPLESECRETNAME: secretValue1
TEST1: secretValue2
_0_TEST_SECRET: secretValue3
_PROD_EXAMPLE_SECRET: secretValue4
SECRET_ALIAS_1: secretValue5
SECRET_ALIAS_2: secretValue6
SECRET2: secretValue7
```

Example 2 Get all secrets that begin with a prefix

The following example creates environment variables for all secrets with names that begin with *beta*.

```
- name: Get Secret Names by Prefix
  uses: 2
  with:
    secret-ids: |
      beta*    # Retrieves all secrets that start with 'beta'
```

Environment variables created:

```
BETASECRETNAME: secretValue1
BETATEST: secretValue2
BETA_NEWSECRET: secretValue3
```

Example 3 Parse JSON in secret

The following example creates environment variables by parsing the JSON in the secret.

```
- name: Get Secrets by Name and by ARN
```

```
  uses: aws-actions/aws-secretsmanager-get-secrets@v2
  with:
    secret-ids: |
      test/secret
      ,secret2
    parse-json-secrets: true
```

The secret `test/secret` has the following secret value.

```
{
  "api_user": "user",
  "api_key": "key",
  "config": {
    "active": "true"
  }
}
```

The secret `secret2` has the following secret value.

```
{
  "myusername": "alejandro_rosalez",
  "mypassword": "EXAMPLE_PASSWORD"
}
```

Environment variables created:

```
TEST_SECRET_API_USER: "user"
TEST_SECRET_API_KEY: "key"
TEST_SECRET_CONFIG_ACTIVE: "true"
MYUSERNAME: "alejandro_rosalez"
MYPASSWORD: "EXAMPLE_PASSWORD"
```

Example 4 Use lowercase letters for environment variable names

The following example creates an environment variable with a lowercase name.

```
- name: Get secrets
  uses: aws-actions/aws-secretsmanager-get-secrets@v2
  with:
    secret-ids: exampleSecretName
```

```
    name-transformation: lowercase
```

Environment variable created:

```
examplesecretname: secretValue
```

Use AWS Secrets Manager in GitLab

AWS Secrets Manager integrates with GitLab. You can leverage Secrets Manager secrets to protect your GitLab credentials so they are no longer hardcoded in GitLab. Instead, GitLab Runner retrieves these secrets from Secrets Manager when your application runs a job in the GitLab CI/CD pipelines.

To use this integration, you'll create an OpenID Connect (OIDC) identity provider in IAM AWS Identity and Access Management and an IAM role. This allows GitLab Runner to access your Secrets Manager secret. For more information about GitLab CI/CD and OIDC, see GitLab documentation.

Considerations

If you're using a non-public GitLab instance, you cannot use this Secrets Manager integration. Instead, see GitLab documentation for non-public instances.

Prerequisites

To integrate Secrets Manager with GitLab, complete the following prerequisites:

1. **Create an AWS Secrets Manager secret**

   You'll need an Secrets Manager secret which will be retrieved in your GitLab job and removes the need to hard-code these credentials. You'll need the Secrets Manager secret ID when you configure your GitLab pipeline. See Create an AWS Secrets Manager secret for more information.
2. **Make GitLab your OIDC provider in the IAM console.**

   In this step, you'll make GitLab your OIDC provider in the IAM console. For more information, see Create an OpenID Connect (OIDC) identity provider and GitLab documentation.

   When creating the OIDC provider in the IAM console, use the following configurations:

   a.

Set the `provider URL` to your GitLab instance. For example, **`gitlab.example.com`**.

b.

Set the `audience` or aud to **`sts.amazonaws.com`**.

3. **Create an IAM role and policy**

   You'll need to create an IAM role and policy. This role is assumed by GitLab with AWS Security Token Service (STS). See Create a role using custom trust policies for more information.

   a. In the IAM console, use the following settings when creating the IAM role:

      - Set `Trusted entity type` to **`Web identity`**.
      - Set `Group` to **`your GitLab group`**.
      - Set `Identity provider` to the same provider URL (the GitLab instance) you used in step 2.
      - Set `Audience` to the same audience you used in step 2.

   b. The following is an example of a trust policy that allows GitLab to assume roles. Your trust policy should list your AWS account, GitLab URL, and project path.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Principal": {
        "Federated": "arn:aws:iam::111122223333:oidc-
provider/gitlab.example.com"
      },
      "Condition": {
        "StringEquals": {
          "gitlab.example.com:aud": [
            "sts.amazon.com"
          ]
        },
        "StringLike": {
          "gitlab.example.com:sub": [
            "project_path:mygroup/project-*:ref_type:branch-*:ref:main*"
          ]
        }
```

```
      }
    }
  ]
}
```

c. You'll also need to create an IAM policy to allow GitLab access to AWS Secrets Manager. You can add this policy to your trust policy. For more information, see Create IAM policies.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "secretsmanager:GetSecretValue",
      "Resource": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:your-secret"
    }
  ]
}
```

Integrating AWS Secrets Manager with GitLab

After completing the prerequisites, you can configure GitLab to use Secrets Manager to protect your credentials.

Configure GitLab pipeline to use Secrets Manager

You'll need to update your GitLab CI/CD configuration file with the following information:

- The audience of the token set to STS.
- The Secrets Manager secret ID.
- The IAM role you want GitLab Runner to assume when executing jobs in the GitLab pipeline.
- The AWS Region where the secret is stored.

GitLab fetches the secret from Secrets Manager and stores the value in a temporary file. The path to this file is stored in a CI/CD variable, similar to file type CI/CD variables.

The following is a snippet of the YAML file for a GitLab CI/CD configuration file:

```
variables:
  AWS_REGION: us-east-1
  AWS_ROLE_ARN: 'arn:aws:iam::111122223333:role/gitlab-role'
job:
  id_tokens:
    AWS_ID_TOKEN:
      aud: 'sts.amazonaws.com'
  secrets:
    DATABASE_PASSWORD:
      aws_secrets_manager:
        secret_id: "arn:aws:secretsmanager:us-east-1:111122223333:secret:secret-name"
```

For more information, see GitLab Secrets Manager integration documentation.

Optionally, you can test your OIDC configuration in GitLab. See GitLab documentation for testing OIDC configuration for more information.

Troubleshooting

The following can help you troubleshoot common issues you might encounter when integrating Secrets Manager with GitLab.

GitLab Pipeline issues

If you experience GitLab pipeline issues, ensure the following:

- Your YAML file is properly formatted. For more information, see GitLab documentation.
- Your GitLab pipeline is assuming the correct role, has the appropriate permissions, and access to the correct AWS Secrets Manager secret.

Additional resources

The following resources can help you troubleshoot issues with GitLab and AWS Secrets Manager:

- GitLab OIDC troubleshooting
- Debugging GitLab CI/CD Pipeline
- Troubleshooting

Use AWS Secrets Manager secrets in AWS IoT Greengrass

AWS IoT Greengrass is software that extends cloud capabilities to local devices. This enables devices to collect and analyze data closer to the source of information, react autonomously to local events, and communicate securely with each other on local networks.

AWS IoT Greengrass lets you authenticate with services and applications from AWS IoT Greengrass devices without hard-coding passwords, tokens, or other secrets. You can use AWS Secrets Manager to securely store and manage your secrets in the cloud. AWS IoT Greengrass extends Secrets Manager to AWS IoT Greengrass core devices, so your connectors and Lambda functions can use local secrets to interact with services and applications.

To integrate a secret into a AWS IoT Greengrass group, you create a group resource that references the Secrets Manager secret. This secret resource references the cloud secret by using the associated ARN. To learn how to create, manage, and use secret resources, see [Working with Secret Resources](https://docs.aws.amazon.com/greengrass/latest/developerguide/secrets-using.html) in the AWS IoT Developer Guide.

To deploy secrets to the AWS IoT Greengrass Core, see Deploy secrets to the AWS IoT Greengrass core.

Use AWS Secrets Manager secrets in Parameter Store

AWS Systems Manager Parameter Store provides secure, hierarchical storage for configuration data management and secrets management. You can store data such as passwords, database strings, and license codes as parameter values. However, Parameter Store doesn't provide automatic rotation services for stored secrets. Instead, Parameter Store enables you to store your secret in Secrets Manager, and then reference the secret as a Parameter Store parameter.

When you configure Parameter Store with Secrets Manager, the `secret-id` Parameter Store requires a forward slash (/) before the name-string.

For more information, see Referencing AWS Secrets Manager Secrets from Parameter Store Parameters in the *AWS Systems Manager User Guide*.

Rotate AWS Secrets Manager secrets

Rotation is the process of periodically updating a secret. When you rotate a secret, you update the credentials in both the secret and the database or service. In Secrets Manager, you can set up automatic rotation for your secrets. There are two forms of rotation:

- Managed rotation – For most managed secrets, you use managed rotation, where the service configures and manages rotation for you. Managed rotation doesn't use a Lambda function.
- Rotate Secrets Manager managed external secrets – For secrets held by Secrets Manager partners, you use managed external secrets rotation to update the secret on the partner's system. This doesn't require a Lambda function.
- the section called "Rotation by Lambda function" – For other types of secrets, Secrets Manager rotation uses a Lambda function to update the secret and the database or service.

Managed rotation for AWS Secrets Manager secrets

Some services offer *managed rotation*, where the service configures and manages rotation for you. With managed rotation, you don't use an AWS Lambda function to update the secret and the credentials in the database.

The following services offer managed rotation:

- **Amazon Aurora** offers managed rotation for master user credentials. For more information, see Password management with Amazon Aurora and AWS Secrets Manager in the *Amazon Aurora User Guide*.
- **Amazon ECS** Service Connect offers managed rotation for AWS Private Certificate Authority TLS certificates. For more information, see TLS with Service Connect in the *Amazon Elastic Container Service Developer Guide*.
- **Amazon RDS** offers managed rotation for master user credentials. For more information, see Password management with Amazon RDS and AWS Secrets Manager in the *Amazon RDS User Guide*.
- **Amazon DocumentDB** offers managed rotation for master user credentials. For more information, see Password management with Amazon DocumentDB and AWS Secrets Manager in the *Amazon DocumentDB User Guide*.

- **Amazon Redshift** offers managed rotation for admin passwords. For more information, see Managing Amazon Redshift admin passwords using AWS Secrets Manager in the *Amazon Redshift Management Guide*.
- **managed external secrets** offers managed rotation for secrets held by Secrets Manager partners. For more information, see Using AWS Secrets Manager managed external secrets to manage Third Party secrets.

> **Tip**
>
> For all other types of secrets, see the section called “Rotation by Lambda function”.

Rotation for managed secrets typically completes within one minute. During rotation, new connections that retrieve the secret may get the previous version of the credentials. In applications, we strongly recommend that you follow the best practice of using a database user created with the minimal privileges required for your application, rather than using the master user. For application users, for highest availability, you can use the Alternating users rotation strategy.

For secrets held by Secrets Manager partners,

To change the schedule for managed rotation

1. Open the managed secret in the Secrets Manager console. You can follow a link from the managing service, or search for the secret in the Secrets Manager console.
2. Under **Rotation schedule**, enter your schedule in UTC time zone in either the **Schedule expression builder** or as a **Schedule expression**. Secrets Manager stores your schedule as a `rate()` or `cron()` expression. The rotation window automatically starts at midnight unless you specify a **Start time**. You can rotate a secret as often as every four hours. For more information, see Rotation schedules.
3. (Optional) For **Window duration**, choose the length of the window during which you want Secrets Manager to rotate your secret, for example **3h** for a three hour window. The window must not extend into the next rotation window. If you don't specify **Window duration**, for a rotation schedule in hours, the window automatically closes after one hour. For a rotation schedule in days, the window automatically closes at the end of the day.
4. Choose **Save**.

To change the schedule for managed rotation (AWS CLI)

- Call `rotate-secret`. The following example rotates the secret between 16:00 and 18:00 UTC on the 1st and 15th day of the month. For more information, see Rotation schedules.

```
aws secretsmanager rotate-secret \
    --secret-id MySecret \
    --rotation-rules \
        "{\"ScheduleExpression\": \"cron(0 16 1,15 * ? *)\", \"Duration\": \"2h\"}"
```

Rotate Secrets Manager managed external secrets

Secrets Manager has partnered with select software vendors to offer managed external secrets. This feature helps customers manage the secret lifecycle by handling rotations automatically. With managed external secrets, customers no longer need to maintain specific rotation logic for each secret stored with different partners. This will be handled by Secrets Manager.

To view the list of partners onboarded with Secrets Manager, see Managed external secrets Partners.

Set Up Rotation in the Console

To configure rotation for an existing managed external secret, created by specifying the secret type and value as specified by the respective integration partners, use the following steps:

1. Open the Secrets Manager console.
2. Select your managed external secret from the list.
3. Choose the **Configuration** tab.
4. In the **Rotation configuration** section, choose **Edit rotation**.
5. Turn on **Automatic rotation**.
6. Under **Rotation metadata**, add any partner-specific metadata required for rotation:

   Follow the guidelines provided by your integration partner for other required metadata
7. In **Service permissions for secret rotation**, select or create an IAM role for rotation:
   - Choose **Create a new role** to automatically create a role with necessary permissions
   - Or select an existing role with appropriate permissions for your partner

By default, permissions are scoped to the individual partner in the region where the secret is created

8. Set your **Rotation schedule** (for example, rotate automatically every 30 days).
9. Choose **Save** to apply the rotation configuration.

The two key metadata fields configured during this process are:

Field	Description
ExternalSecretRotationMetadata	Partner-specific metadata required for rotation, such as API version for Salesforce
ExternalSecretRotationRoleArn	The ARN of the IAM role used for rotation, with permissions scoped to the integration partner

For more information on these fields, see Using Secrets Manager managed external secrets to manage Third Party secrets.

Set Up Rotation Using the CLI

Run the following command to set up rotation for a Salesforce secret. This command specifies the secret ID, the IAM role ARN for rotation, the rotation schedule, and any partner-specific metadata required for the rotation process.

```
aws secretsmanager rotate-secret \
            --secret-id SampleSecret \
            --external-secret-rotation-role-arn arn:aws:iam::123412341234:role/xyz \
            --rotation-rules AutomaticallyAfterDays=1 \
            --external-secret-rotation-metadata
 '[{"Key":"apiVersion","Value":"v65.0"}]'
```

Rotation by Lambda function

For many types of secrets, Secrets Manager uses an AWS Lambda function to update the secret and the database or service. For information about the costs of using a Lambda function, see Pricing.

For some Secrets managed by other services, you use *managed rotation*. To use Managed rotation, you first create the secret through the managing service.

During rotation, Secrets Manager logs events that indicate the state of rotation. For more information, see the section called "Log with AWS CloudTrail ".

To rotate a secret, Secrets Manager calls a Lambda function according to the rotation schedule you set up. If you also manually update your secret value while automatic rotation is set up, then Secrets Manager considers that a valid rotation when it calculates the next rotation date.

During rotation, Secrets Manager calls the same function several times, each time with different parameters. Secrets Manager invokes the function with the following JSON request structure of parameters:

```
{
    "Step" : "request.type",
    "SecretId" : "string",
    "ClientRequestToken" : "string",
    "RotationToken" : "string"
}
```

Parameters:

- **Step** – The rotation step: `create_secret`, `set_secret`, `test_secret`, or `finish_secret`. For more information, see the section called "Four steps in a rotation function".
- **SecretId** – The ARN of the secret to rotate.
- **ClientRequestToken** – A unique identifier for the new version of the secret. This value helps ensure idempotency. For more information, see PutSecretValue: ClientRequestToken in the *AWS Secrets Manager API Reference*.
- **RotationToken** – A unique identifier that indicates the source of the request. Required for secret rotation using an assumed role or cross-account rotation, in which you rotate a secret in one account by using a Lambda rotation function in another account. In both cases, the rotation function assumes an IAM role to call Secrets Manager and then Secrets Manager uses the rotation token to validate the IAM role identity.

If any rotation step fails, Secrets Manager retries the entire rotation process multiple times.

Topics

- Set up automatic rotation for Amazon RDS, Amazon Aurora, Amazon Redshift, or Amazon DocumentDB secrets
- Set up automatic rotation for non-database AWS Secrets Manager secrets
- Set up automatic rotation using the AWS CLI
- Lambda function rotation strategies
- Lambda rotation functions
- AWS Secrets Manager rotation function templates
- Lambda rotation function execution role permissions for AWS Secrets Manager
- Network access for AWS Lambda rotation function
- Troubleshoot AWS Secrets Manager rotation

Set up automatic rotation for Amazon RDS, Amazon Aurora, Amazon Redshift, or Amazon DocumentDB secrets

This tutorial describes how to set up the section called “Rotation by Lambda function” for database secrets. Rotation is the process of periodically updating a secret. When you rotate a secret, you update the credentials in both the secret and the database. In Secrets Manager, you can set up automatic rotation for your database secrets.

To set up rotation using the console, you need to first choose a rotation strategy. Then you configure the secret for rotation, which creates a Lambda rotation function if you don't already have one. The console also sets permissions for the Lambda function execution role. The last step is to make sure that the Lambda rotation function can access both Secrets Manager and your database through the network.

> **Warning**
>
> To turn on automatic rotation, you must have permission to create an IAM execution role for the Lambda rotation function and attach a permission policy to it. You need both `iam:CreateRole` and `iam:AttachRolePolicy` permissions. Granting these permissions allows an identity to grant themselves any permissions.

Steps:

- Step 1: Choose a rotation strategy and (optionally) create a superuser secret

- Step 2: Configure rotation and create a rotation function
- Step 3: (Optional) Set additional permissions conditions on the rotation function
- Step 4: Set up network access for the rotation function
- Next steps

Step 1: Choose a rotation strategy and (optionally) create a superuser secret

For information about the strategies offered by Secrets Manager, see the section called “Lambda function rotation strategies”.

If you choose the *alternating users strategy*, you must Create secrets and store database superuser credentials in it. You need a secret with superuser credentials because rotation clones the first user, and most users do not have that permission. Note that Amazon RDS Proxy does not support the alternating users strategy.

Step 2: Configure rotation and create a rotation function

To turn on rotation for an Amazon RDS, Amazon DocumentDB, or Amazon Redshift secret

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. On the **Secrets** page, choose your secret.
3. On the **Secret details** page, in the **Rotation configuration** section, choose **Edit rotation**.
4. In the **Edit rotation configuration** dialog box, do the following:

    a. Turn on **Automatic rotation**.

    b. Under **Rotation schedule**, enter your schedule in UTC time zone in either the **Schedule expression builder** or as a **Schedule expression**. Secrets Manager stores your schedule as a `rate()` or `cron()` expression. The rotation window automatically starts at midnight unless you specify a **Start time**. You can rotate a secret as often as every four hours. For more information, see Rotation schedules.

    c. (Optional) For **Window duration**, choose the length of the window during which you want Secrets Manager to rotate your secret, for example **3h** for a three hour window. The window must not extend into the next rotation window. If you don't specify **Window duration**, for a rotation schedule in hours, the window automatically closes after one hour. For a rotation schedule in days, the window automatically closes at the end of the day.

d. (Optional) Choose **Rotate immediately when the secret is stored** to rotate your secret when you save your changes. If you clear the checkbox, then the first rotation will begin on the schedule you set.

   If rotation fails, for example because Steps 3 and 4 are not yet completed, Secrets Manager retries the rotation process multiple times.

e. Under **Rotation function**, do one of the following:

   - Choose **Create a new Lambda function** and enter a name for your new function. Secrets Manager adds `SecretsManager` to the beginning of the function name. Secrets Manager creates the function based on the appropriate [template] and sets the necessary [permissions] for the Lambda execution role.
   - Choose **Use an existing Lambda function** to reuse a rotation function you used for another secret. The rotation functions listed under **Recommended VPC configurations** have the same VPC and security group as the database, which helps the function access the database.

f. For **Rotation strategy**, choose the **Single user** or **Alternating users** strategy. For more information, see the section called "Step 1: Choose a rotation strategy and (optionally) create a superuser secret".

5. Choose **Save**.

Step 3: (Optional) Set additional permissions conditions on the rotation function

In the resource policy for your rotation function, we recommend that you include the context key `aws:SourceAccount` to help prevent Lambda from being used as a confused deputy. For some AWS services, to avoid the confused deputy scenario, AWS recommends that you use both the `aws:SourceArn` and `aws:SourceAccount` global condition keys. However, if you include the `aws:SourceArn` condition in your rotation function policy, the rotation function can only be used to rotate the secret specified by that ARN. We recommend that you include only the context key `aws:SourceAccount` so that you can use the rotation function for multiple secrets.

To update your rotation function resource policy

1. In the Secrets Manager console, choose your secret, and then on the details page, under **Rotation configuration**, choose the Lambda rotation function. The Lambda console opens.
2. Follow the instructions at Using resource-based policies for Lambda to add a `aws:sourceAccount` condition.

```
"Condition": {
    "StringEquals": {
        "AWS:SourceAccount": "123456789012"
    }
},
```

If the secret is encrypted with a KMS key other than the AWS managed key `aws/secretsmanager`, Secrets Manager grants the Lambda execution role permission to use the key. You can use the SecretARN encryption context to limit the use of the decrypt function, so the rotation function role only has access to decrypt the secret it is responsible for rotating.

To update your rotation function execution role

1. From the Lambda rotation function, choose **Configuration**, and then under **Execution role**, choose the **Role name**.
2. Follow the instructions at Modifying a role permissions policy to add a `kms:EncryptionContext:SecretARN` condition.

```
"Condition": {
    "StringEquals": {
        "kms:EncryptionContext:SecretARN": "SecretARN"
    }
},
```

Step 4: Set up network access for the rotation function

For more information, see the section called "Network access for AWS Lambda rotation function".

Next steps

See the section called "Troubleshoot rotation".

Set up automatic rotation for non-database AWS Secrets Manager secrets

This tutorial describes how to set up the section called "Rotation by Lambda function" for non-database secrets. Rotation is the process of periodically updating a secret. When you rotate a secret, you update the credentials in both the secret and the database or service that the secret is for.

For database secrets, see Automatic rotation for database secrets (console).

> **Warning**
>
> To turn on automatic rotation, you must have permission to create an IAM execution role for the Lambda rotation function and attach a permission policy to it. You need both `iam:CreateRole` and `iam:AttachRolePolicy` permissions. Granting these permissions allows an identity to grant themselves any permissions.

Steps:

Step 1: Create a generic rotation function

To begin, create a Lambda rotation function. It will not have the code in it to rotate your secret, so you'll write that in a later step. For information about how a rotation function works, see the section called "Lambda rotation functions".

In supported Regions, you can use AWS Serverless Application Repository to create the function from a template. For a list of supported Regions, see AWS Serverless Application Repository FAQs. In other Regions, you create the function from scratch and copy the template code into the function.

To create a generic rotation function

1. To determine whether AWS Serverless Application Repository is supported in your Region, see AWS Serverless Application Repository endpoints and quotas in the *AWS General Reference*.

2. Do one of the following:

   - If AWS Serverless Application Repository is supported in your Region:

     a. In the Lambda console, choose **Applications** and then choose **Create application**.

     b. On the **Create application** page, choose the **Serverless application** tab.

     c. In the search box under **Public applications**, enter `SecretsManagerRotationTemplate`.

     d. Select **Show apps that create custom IAM roles or resource policies**.

     e. Choose the **SecretsManagerRotationTemplate** tile.

     f. On the **Review, configure and deploy** page, in the **Application settings** tile, fill in the required fields.

        - For **endpoint**, enter the endpoint for your Region, including `https://`. For a list of endpoints, see the section called "Secrets Manager endpoints".
        - To put the Lambda function in a VPC, include **vpcSecurityGroupIds** and **vpcSubnetIds**.

     g. Choose **Deploy**.

   - If AWS Serverless Application Repository isn't supported in your Region:

     a. In the Lambda console, choose **Functions** and then choose **Create function**.

     b. On the **Create function** page, do the following:

        i. Choose **Author from scratch**.

        ii. For **Function name**, enter a name for your rotation function.

        iii. For **Runtime**, choose **Python 3.10**.

        iv. Choose **Create function**.

Step 2: Write the rotation function code

In this step, you write the code that updates the secret and the service or database that the secret is for. For information about what a rotation function does, including tips on writing your own rotation function, see the section called "Lambda rotation functions". You can also use the Rotation function templates as reference.

Step 3: Configure the secret for rotation

In this step, you set a rotation schedule for your secret and connect the rotation function to the secret.

To configure rotation and create an empty rotation function

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. On the **Secrets** page, choose your secret.
3. On the **Secret details** page, in the **Rotation configuration** section, choose **Edit rotation**. In the **Edit rotation configuration** dialog box, do the following:

   a. Turn on **Automatic rotation**.

   b. Under **Rotation schedule**, enter your schedule in UTC time zone in either the **Schedule expression builder** or as a **Schedule expression**. Secrets Manager stores your schedule as a `rate()` or `cron()` expression. The rotation window automatically starts at midnight unless you specify a **Start time**. You can rotate a secret as often as every four hours. For more information, see Rotation schedules.

   c. (Optional) For **Window duration**, choose the length of the window during which you want Secrets Manager to rotate your secret, for example **3h** for a three hour window. The window must not extend into the next rotation window. If you don't specify **Window duration**, for a rotation schedule in hours, the window automatically closes after one hour. For a rotation schedule in days, the window automatically closes at the end of the day.

   d. (Optional) Choose **Rotate immediately when the secret is stored** to rotate your secret when you save your changes. If you clear the checkbox, then the first rotation will begin on the schedule you set.

   e. Under **Rotation function**, choose the Lambda function you created in Step 1.

   f. Choose **Save**.

Step 4: Allow the rotation function to access Secrets Manager and your database or service

The Lambda rotation function needs permission to access the secret in Secrets Manager, and it needs permission to access your database or service. In this step, you grant these permissions to the Lambda execution role. If the secret is encrypted with a KMS key other than the AWS managed key `aws/secretsmanager`, then you need to grant the Lambda execution role permission to use the key. You can use the SecretARN encryption context to limit the use of the decrypt function, so the rotation function role only has access to decrypt the secret it is responsible for rotating. For policy examples, see Permissions for rotation.

For instructions, see Lambda execution role in the *AWS Lambda Developer Guide*.

Step 5: Allow Secrets Manager to invoke the rotation function

To allow Secrets Manager to invoke the rotation function on the rotation schedule you set up, you need to grant `lambda:InvokeFunction` permission to the Secrets Manager service principal in the resource policy of the Lambda function.

In the resource policy for your rotation function, we recommend that you include the context key `aws:SourceAccount` to help prevent Lambda from being used as a confused deputy. For some AWS services, to avoid the confused deputy scenario, AWS recommends that you use both the `aws:SourceArn` and `aws:SourceAccount` global condition keys. However, if you include the `aws:SourceArn` condition in your rotation function policy, the rotation function can only be used to rotate the secret specified by that ARN. We recommend that you include only the context key `aws:SourceAccount` so that you can use the rotation function for multiple secrets.

To attach a resource policy to a Lambda function, see Using resource-based policies for Lambda.

The following policy allows Secrets Manager to invoke a Lambda function.

JSON

```
{
    "Version":"2012-10-17",
    "Id": "default",
    "Statement": [
    {
        "Effect": "Allow",
        "Principal": {
            "Service": "secretsmanager.amazonaws.com"
            },
        "Action": "lambda:InvokeFunction",
        "Condition": {
            "StringEquals": {
                "AWS:SourceAccount": "123456789012"
            }
```

```
        },
        "Resource": "arn:aws:lambda:us-east-1:123456789012:function:function-
name"
    }
    ]
}
```

Step 6: Set up network access for the rotation function

In this step, you allow the rotation function to connect to both Secrets Manager and the service or database the secret is for. The rotation function must have access to both to be able to rotate the secret. See the section called "Network access for AWS Lambda rotation function".

Next steps

When you configured rotation in Step 3, you set a schedule for rotating the secret. If rotation fails when it is scheduled, Secrets Manager will attempt the rotation multiple times. You can also start a rotation immediately by following the instructions in Rotate a secret immediately.

If rotation fails, see Troubleshoot rotation.

Set up automatic rotation using the AWS CLI

This tutorial describes how to set up the section called "Rotation by Lambda function" by using the AWS CLI. When you rotate a secret, you update the credentials in both the secret and the database or service that the secret is for.

You can also set up rotation using the console. For database secrets, see Automatic rotation for database secrets (console). For all other types of secrets, see Automatic rotation for non-database secrets (console).

To set up rotation using the AWS CLI, if you are rotating a database secret, you first need to choose a rotation strategy. If you choose the alternating users strategy, you must store a separate secret with credentials for a database superuser. Next, you write the rotation function code. Secrets Manager provides templates you can base your function on. Then you create a Lambda function with your code and set permissions for both the Lambda function and the Lambda execution role. The next step is to make sure that the Lambda function can access both Secrets Manager and your database or service through the network. Finally, you configure the secret for rotation.

Steps:

Prerequisite for database secrets: Choose a rotation strategy

For information about the strategies offered by Secrets Manager, see the section called "Lambda function rotation strategies".

Option 1: Single user strategy

If you choose the *single user strategy*, you can continue with Step 1.

Option 2: Alternating users strategy

If you choose the *alternating users strategy*, you must:

- Create a secret and store database superuser credentials in it. You need a secret with superuser credentials because alternating users rotation clones the first user, and most users do not have that permission.
- Add the ARN of the superuser secret to the original secret. For more information, see the section called "JSON structure of a secret".

Note that Amazon RDS Proxy does not support the alternating users strategy.

Step 1: Write the rotation function code

To rotate a secret, you need a rotation function. A rotation function is a Lambda function that Secrets Manager calls to rotate your secret. For more information, see the section called "Rotation by Lambda function". In this step, you write the code that updates the secret and the service or database that the secret is for.

Secrets Manager provides templates for Amazon RDS, Amazon Aurora, Amazon Redshift, and Amazon DocumentDB database secrets in Rotation function templates.

To write the rotation function code

1. Do one of the following:
   - Check the list of rotation function templates. If there is one that matches your service and rotation strategy, copy the code.
   - For other types of secrets, you write your own rotation function. For instructions, see the section called "Lambda rotation functions".
2. Save the file in a ZIP file *my-function.zip* along with any required dependencies.

Step 2: Create the Lambda function

In this step, you create the Lambda function using the ZIP file you created in Step 1. You also set the Lambda execution role, which is the role that Lambda assumes when the function is invoked.

To create a Lambda rotation function and execution role

1. Create a trust policy for the Lambda execution role and save it as a JSON file. For examples and more information, see the section called "Permissions for rotation". The policy must:
   - Allow the role to call Secrets Manager operations on the secret.
   - Allow the role to call the service that the secret is for, for example, to create a new password.
2. Create the Lambda execution role and apply the trust policy you created in the previous step by calling `iam create-role`.

```
aws iam create-role \
    --role-name rotation-lambda-role \
    --assume-role-policy-document file://trust-policy.json
```

3. Create the Lambda function from the ZIP file by calling `lambda create-function`.

```
aws lambda create-function \
  --function-name my-rotation-function \
  --runtime python3.7 \
  --zip-file fileb://my-function.zip \
  --handler .handler \
  --role arn:aws:iam::123456789012:role/service-role/rotation-lambda-role
```

4. Set a resource policy on the Lambda function to allow Secrets Manager to invoke it by calling `lambda add-permission`.

```
aws lambda add-permission \
  --function-name my-rotation-function \
  --action lambda:InvokeFunction \
  --statement-id SecretsManager \
  --principal secretsmanager.amazonaws.com \
  --source-account 123456789012
```

Step 3: Set up network access

For more information, see the section called “Network access for AWS Lambda rotation function”.

Step 4: Configure the secret for rotation

To turn on automatic rotation for your secret, call `rotate-secret`. You can set a rotation schedule with a `cron()` or `rate()` schedule expression, and you can set a rotation window duration. For more information, see the section called “Rotation schedules”.

```
aws secretsmanager rotate-secret \
    --secret-id MySecret \
    --rotation-lambda-arn arn:aws:lambda:Region:123456789012:function:my-rotation-function \
    --rotation-rules "{\"ScheduleExpression\": \"cron(0 16 1,15 * ? *)\", \"Duration\": \"2h\"}"
```

Next steps

See the section called “Troubleshoot rotation”.

Lambda function rotation strategies

For the section called “Rotation by Lambda function”, for database secrets, Secrets Manager offers two rotation strategies.

Rotation strategy: single user

This strategy updates credentials for one user in one secret. For Amazon RDS Db2 instances, because users can't change their own passwords, you must provide admin credentials in a separate secret. **This is the simplest rotation strategy, and it is appropriate for most use cases.** In particular, we recommend you use this strategy for credentials for one-time (ad hoc) or interactive users.

When the secret rotates, open database connections are not dropped. While rotation is happening, there is a short period of time between when the password in the database changes and when the secret is updated. During this time, there is a low risk of the database denying calls that use the rotated credentials. You can mitigate this risk with an appropriate retry strategy. After rotation, new connections use the new credentials.

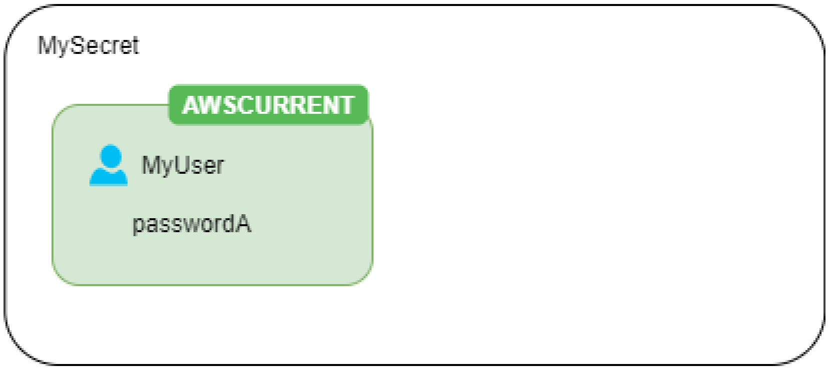
Rotation strategy: alternating users

This strategy updates credentials for two users in one secret. You create the first user, and during the first rotation, the rotation function clones it to create the second user. Every time the secret rotates, the rotation function alternates which user's password it updates. Because most users don't have permission to clone themselves, you must provide the credentials for a `superuser` in another secret. We recommend using the single-user rotation strategy when cloned users in your database don't have the same permissions as the original user, and for credentials for one-time (ad hoc) or interactive users.

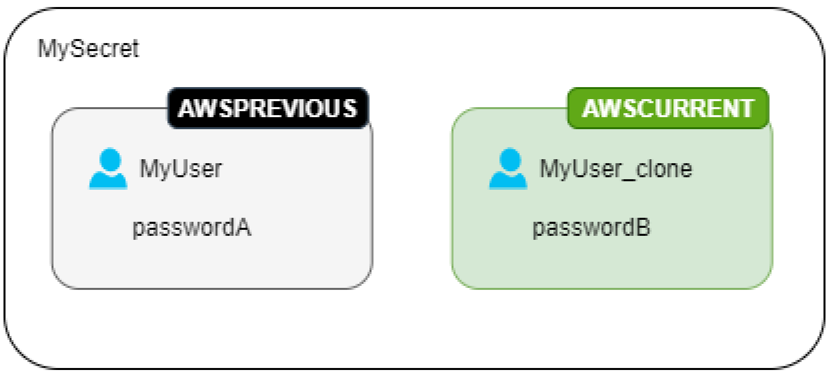
This strategy is appropriate for databases with permission models where one role owns the database tables and a second role has permission to access the database tables. It is also appropriate for applications that require high availability. If an application retrieves the secret during rotation, the application still gets a valid set of credentials. After rotation, both `user` and `user_clone` credentials are valid. There is even less chance of applications getting a deny during this type of rotation than single user rotation. If the database is hosted on a server farm where the password change takes time to propagate to all servers, there is a risk of the database denying calls that use the new credentials. You can mitigate this risk with an appropriate retry strategy.

Secrets Manager creates the cloned user with the same permissions as the original user. If you change the original user's permissions after the clone is created, you must also change the cloned user's permissions.

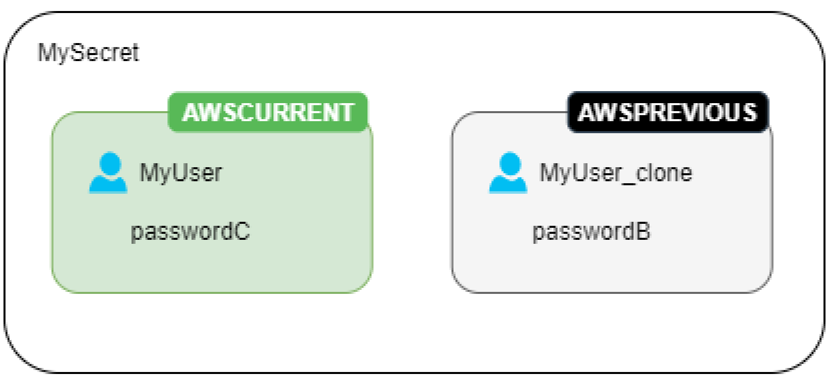
For example, if you create a secret with a database user's credentials, the secret contains one version with those credentials.



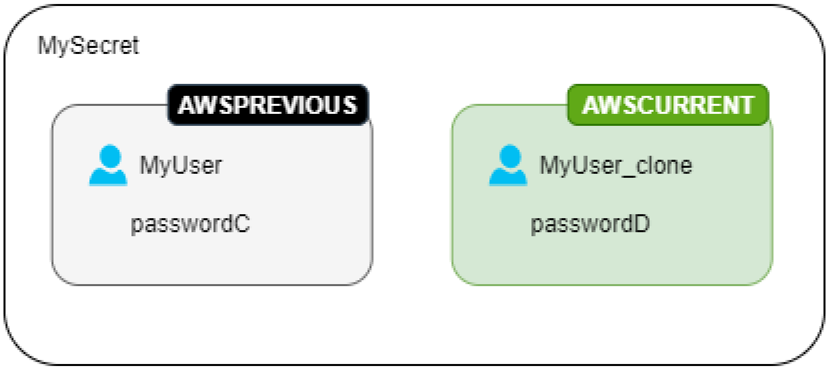
First rotation – The rotation function creates a clone of your user with a generated password, and those credentials become the current secret version.



Second rotation – The rotation function updates the password for the original user.



Third rotation – The rotation function updates the password for the cloned user.



Lambda rotation functions

In the section called "Rotation by Lambda function", an AWS Lambda function rotates the secret. AWS Secrets Manager uses staging labels to identify secret versions during rotation.

If AWS Secrets Manager doesn't provide a rotation function template for your secret type, you can create a custom rotation function. Follow these guidelines when writing your rotation function:

Best practices for custom rotation functions

- Use the generic rotation template as a starting point.
- Be cautious with debugging or logging statements. They can write information to Amazon CloudWatch Logs. Ensure logs don't contain sensitive information.

  For log statement examples, see the the section called "Rotation function templates" source code.
- For security, AWS Secrets Manager only allows a Lambda rotation function to rotate the secret directly. The rotation function can't call another Lambda function to rotate the secret.
- For debugging guidance, see Testing and debugging serverless applications.
- If you use external binaries and libraries, for example to connect to a resource, you're responsible for patching and updating them.
- Package your rotation function and any dependencies in a ZIP file, such as *`my-function.zip`*.

Warning

Setting the provisioned concurrency parameter to a value lower than 10 can cause throttling due to insufficient execution threads for the Lambda function. For more information, see [Understanding reserved concurrency and provisioned concurrency] in the AWS Lambda AWS Lambda Developer Guide.

Four steps in a rotation function

Topics

- createSecret: Create a new version of the secret
- setSecret: Change the credentials in the database or service
- testSecret: Test the new secret version
- finishSecret: Finish the rotation

`createSecret`: Create a new version of the secret

The method `createSecret` first checks if a secret exists by calling `get_secret_value` with the passed-in `ClientRequestToken`. If there's no secret, it creates a new secret with `create_secret` and the token as the `VersionId`. Then it generates a new secret value with `get_random_password`. Next it calls `put_secret_value` to store it with the staging label `AWSPENDING`. Storing the new secret value in `AWSPENDING` helps ensure idempotency. If rotation fails for any reason, you can refer to that secret value in subsequent calls. See How do I make my Lambda function idempotent.

Tips for writing your own rotation function

- Ensure the new secret value only includes characters that are valid for the database or service. Exclude characters by using the `ExcludeCharacters` parameter.
- As you test your function, use the AWS CLI to see version stages: call `describe-secret` and look at `VersionIdsToStages`.
- For Amazon RDS MySQL, in alternating users rotation, Secrets Manager creates a cloned user with a name no longer than 16 characters. You can modify the rotation function to allow longer usernames. MySQL version 5.7 and higher supports usernames up to 32 characters, however Secrets Manager appends "_clone" (six characters) to the end of the username, so you must keep the username to a maximum of 26 characters.

setSecret: Change the credentials in the database or service

The method `setSecret` changes the credential in the database or service to match the new secret value in the `AWSPENDING` version of the secret.

Tips for writing your own rotation function

- If you pass statements to a service that interprets statements, like a database, use query parameterization. For more information, see [Query Parameterization Cheat Sheet]() on the *OWASP web site*.
- The rotation function is a privileged deputy that has the authorization to access and modify customer credentials in both the Secrets Manager secret and the target resource. To prevent a potential [confused deputy attack](), you need to make sure that an attacker cannot use the function to access other resources. Before you update the credential:
  - Check that the credential in the `AWSCURRENT` version of the secret is valid. If the `AWSCURRENT` credential isn't valid, abandon the rotation attempt.
  - Check that the `AWSCURRENT` and `AWSPENDING` secret values are for the same resource. For a username and password, check that the `AWSCURRENT` and `AWSPENDING` usernames are the same.
  - Check that the destination service resource is the same. For a database, check that the `AWSCURRENT` and `AWSPENDING` host names are the same.
- In rare cases, you might want to customize an existing rotation function for a database. For example, with alternating users rotation, Secrets Manager creates the cloned user by copying the [runtime configuration parameters]() of the first user. If you want to include more attributes, or change which ones are granted to the cloned user, you need to update the code in the `set_secret` function.

testSecret: Test the new secret version

Next, the Lambda rotation function tests the `AWSPENDING` version of the secret by using it to access the database or service. Rotation functions based on [Rotation function templates]() test the new secret by using read access.

finishSecret: Finish the rotation

Finally, the Lambda rotation function moves the label `AWSCURRENT` from the previous secret version to this version, which also removes the `AWSPENDING` label in the same API call. Secrets

Manager adds the `AWSPREVIOUS` staging label to the previous version, so that you retain the last known good version of the secret.

The method **finish_secret** uses `update_secret_version_stage` to move the staging label `AWSCURRENT` from the previous secret version to the new secret version. Secrets Manager automatically adds the `AWSPREVIOUS` staging label to the previous version, so that you retain the last known good version of the secret.

Tips for writing your own rotation function

- Don't remove `AWSPENDING` before this point, and don't remove it by using a separate API call, because that can indicate to Secrets Manager that the rotation did not complete successfully. Secrets Manager adds the `AWSPREVIOUS` staging label to the previous version, so that you retain the last known good version of the secret.

When rotation is successful, the `AWSPENDING` staging label might be attached to the same version as the `AWSCURRENT` version, or it might not be attached to any version. If the `AWSPENDING` staging label is present but not attached to the same version as `AWSCURRENT`, then any later invocation of rotation assumes that a previous rotation request is still in progress and returns an error. When rotation is unsuccessful, the `AWSPENDING` staging label might be attached to an empty secret version. For more information, see Troubleshoot rotation.

AWS Secrets Manager rotation function templates

AWS Secrets Manager provides a set of rotation function templates that help automate the secure management of credentials for various database systems and services. The templates are ready-to-use Lambda functions that implement best practices for credential rotation, helping you maintain your security posture without manual intervention.

The templates support two primary rotation strategies:

- *Single-user rotation* which updates the credentials for a single user.
- *Alternating-users rotation* which maintains two separate users to help eliminate downtime during credential changes.

Secrets Manager also provides a generic template that serves as a starting point for any type of secret.

To use the templates, see:

- Automatic rotation for database secrets (console)
- Automatic rotation for non-database secrets (console)

To write your own rotation function, see Write a rotation function.

Templates

- Amazon RDS and Amazon Aurora
  - Amazon RDS Db2 single user
  - Amazon RDS Db2 alternating users
  - Amazon RDS MariaDB single user
  - Amazon RDS MariaDB alternating users
  - Amazon RDS and Amazon Aurora MySQL single user
  - Amazon RDS and Amazon Aurora MySQL alternating users
  - Amazon RDS Oracle single user
  - Amazon RDS Oracle alternating users
  - Amazon RDS and Amazon Aurora PostgreSQL single user
  - Amazon RDS and Amazon Aurora PostgreSQL alternating users
  - Amazon RDS Microsoft SQLServer single user
  - Amazon RDS Microsoft SQLServer alternating users
- Amazon DocumentDB (with MongoDB compatibility)
  - Amazon DocumentDB single user
  - Amazon DocumentDB alternating users
- Amazon Redshift
  - Amazon Redshift single user
  - Amazon Redshift alternating users
- Amazon Timestream for InfluxDB
  - Amazon Timestream for InfluxDB single user
  - Amazon Timestream for InfluxDB alternating users
- Amazon ElastiCache
- Active Directory
  - Active Directory credentials

  - Active Directory keytab
- Other types of secrets

Amazon RDS and Amazon Aurora

Amazon RDS Db2 single user

- **Template name:** SecretsManagerRDSDb2RotationSingleUser
- **Rotation strategy:** Rotation strategy: single user.
- **`SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSDb2RotationSingleUser/lambda_function.py
- **Dependency:** python-ibmdb

Amazon RDS Db2 alternating users

- **Template name:** SecretsManagerRDSDb2RotationMultiUser
- **Rotation strategy:** the section called "Alternating users".
- **`SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSDb2RotationMultiUser/lambda_function.py
- **Dependency:** python-ibmdb

Amazon RDS MariaDB single user

- **Template name:** SecretsManagerRDSMariaDBRotationSingleUser
- **Rotation strategy:** Rotation strategy: single user.
- **`SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSMariaDBRotationSingleUser/lambda_function.py
- **Dependency:** PyMySQL 1.0.2. If you use sha256 password for authentication, PyMySQL[rsa]. For information about using packages with compiled code in a Lambda runtime, see How do I add Python packages with compiled binaries to my deployment package and make the package compatible with Lambda? in *AWS Knowledge Center*.

Amazon RDS MariaDB alternating users

- **Template name:** SecretsManagerRDSMariaDBRotationMultiUser
- **Rotation strategy:** the section called "Alternating users".
- **`SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSMariaDBRotationMultiUser/lambda_function.py
- **Dependency:** PyMySQL 1.0.2. If you use sha256 password for authentication, PyMySQL[rsa]. For information about using packages with compiled code in a Lambda runtime, see How do I add Python packages with compiled binaries to my deployment package and make the package compatible with Lambda? in *AWS Knowledge Center*.

Amazon RDS and Amazon Aurora MySQL single user

- **Template name:** SecretsManagerRDSMySQLRotationSingleUser
- **Rotation strategy:** the section called "Single user".
- **Expected `SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSMySQLRotationSingleUser/lambda_function.py
- **Dependency:** PyMySQL 1.0.2. If you use sha256 password for authentication, PyMySQL[rsa]. For information about using packages with compiled code in a Lambda runtime, see How do I add Python packages with compiled binaries to my deployment package and make the package compatible with Lambda? in *AWS Knowledge Center*.

Amazon RDS and Amazon Aurora MySQL alternating users

- **Template name:** SecretsManagerRDSMySQLRotationMultiUser
- **Rotation strategy:** the section called "Alternating users".
- **Expected `SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSMySQLRotationMultiUser/lambda_function.py
- **Dependency:** PyMySQL 1.0.2. If you use sha256 password for authentication, PyMySQL[rsa]. For information about using packages with compiled code in a Lambda runtime, see How do I

add Python packages with compiled binaries to my deployment package and make the package compatible with Lambda? in *AWS Knowledge Center*.

Amazon RDS Oracle single user

- **Template name:** SecretsManagerRDSOracleRotationSingleUser
- **Rotation strategy:** the section called "Single user".
- **Expected `SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSOracleRotationSingleUser/lambda_function.py
- **Dependency:** python-oracledb 2.4.1

Amazon RDS Oracle alternating users

- **Template name:** SecretsManagerRDSOracleRotationMultiUser
- **Rotation strategy:** the section called "Alternating users".
- **Expected `SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSOracleRotationMultiUser/lambda_function.py
- **Dependency:** python-oracledb 2.4.1

Amazon RDS and Amazon Aurora PostgreSQL single user

- **Template name:** SecretsManagerRDSPostgreSQLRotationSingleUser
- **Rotation strategy:** Rotation strategy: single user.
- **Expected `SecretString` structure:** the section called "Amazon RDS and Aurora credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSPostgreSQLRotationSingleUser/lambda_function.py
- **Dependency:** PyGreSQL 5.2.5

Amazon RDS and Amazon Aurora PostgreSQL alternating users

- **Template name:** SecretsManagerRDSPostgreSQLRotationMultiUser
- **Rotation strategy:** the section called "Alternating users".

- **Expected `SecretString` structure:** the section called “Amazon RDS and Aurora credentials”.
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSPostgreSQLRotationMultiUser/lambda_function.py
- **Dependency:** PyGreSQL 5.2.5

Amazon RDS Microsoft SQLServer single user

- **Template name:** SecretsManagerRDSSQLServerRotationSingleUser
- **Rotation strategy:** the section called “Single user”.
- **Expected `SecretString` structure:** the section called “Amazon RDS and Aurora credentials”.
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSSQLServerRotationSingleUser/lambda_function.py
- **Dependency:** Pymssql 2.2.2

Amazon RDS Microsoft SQLServer alternating users

- **Template name:** SecretsManagerRDSSQLServerRotationMultiUser
- **Rotation strategy:** the section called “Alternating users”.
- **Expected `SecretString` structure:** the section called “Amazon RDS and Aurora credentials”.
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRDSSQLServerRotationMultiUser/lambda_function.py
- **Dependency:** Pymssql 2.2.2

Amazon DocumentDB (with MongoDB compatibility)

Amazon DocumentDB single user

- **Template name:** SecretsManagerMongoDBRotationSingleUser
- **Rotation strategy:** the section called “Single user”.
- **Expected `SecretString` structure:** the section called “Amazon DocumentDB credentials”.
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerMongoDBRotationSingleUser/lambda_function.py
- **Dependency:** PyMongo 4.2.0

Amazon DocumentDB alternating users

- **Template name:** SecretsManagerMongoDBRotationMultiUser
- **Rotation strategy:** the section called "Alternating users".
- **Expected `SecretString` structure:** the section called "Amazon DocumentDB credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerMongoDBRotationMultiUser/lambda_function.py
- **Dependency:** PyMongo 4.2.0

Amazon Redshift

Amazon Redshift single user

- **Template name:** SecretsManagerRedshiftRotationSingleUser
- **Rotation strategy:** the section called "Single user".
- **Expected `SecretString` structure:** the section called "Amazon Redshift credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRedshiftRotationSingleUser/lambda_function.py
- **Dependency:** PyGreSQL 5.2.5

Amazon Redshift alternating users

- **Template name:** SecretsManagerRedshiftRotationMultiUser
- **Rotation strategy:** the section called "Alternating users".
- **Expected `SecretString` structure:** the section called "Amazon Redshift credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRedshiftRotationMultiUser/lambda_function.py
- **Dependency:** PyGreSQL 5.2.5

Amazon Timestream for InfluxDB

To use these templates, see How Amazon Timestream for InfluxDB uses secrets in the *Amazon Timestream Developer Guide*.

Amazon Timestream for InfluxDB single user

- **Template name:** SecretsManagerInfluxDBRotationSingleUser
- **Expected `SecretString` structure:** the section called "Amazon Timestream for InfluxDB secret structure".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerInfluxDBRotationSingleUser/lambda_function.py
- **Dependency:** InfluxDB 2.0 python client

Amazon Timestream for InfluxDB alternating users

- **Template name:** SecretsManagerInfluxDBRotationMultiUser
- **Expected `SecretString` structure:** the section called "Amazon Timestream for InfluxDB secret structure".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerInfluxDBRotationMultiUser/lambda_function.py
- **Dependency:** InfluxDB 2.0 python client

Amazon ElastiCache

To use this template, see Automatically rotating passwords for users in the *Amazon ElastiCache User Guide*.

- **Template name:** SecretsManagerElasticacheUserRotation
- **Expected `SecretString` structure:** the section called "Amazon ElastiCache credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerElasticacheUserRotation/lambda_function.py

Active Directory

Active Directory credentials

- **Template name:** SecretsManagerActiveDirectoryRotationSingleUser
- **Expected `SecretString` structure:** the section called "Active Directory credentials".

- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerActiveDirectoryRotationSingleUser/lambda_function.py

Active Directory keytab

- **Template name:** SecretsManagerActiveDirectoryAndKeytabRotationSingleUser
- **Expected `SecretString` structure:** the section called "Active Directory credentials".
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerActiveDirectoryAndKeytabRotationSingleUser/lambda_function.py
- **Dependencies:** msktutil

Other types of secrets

Secrets Manager provides this template as a starting point for you to create a rotation function for any type of secret.

- **Template name:** SecretsManagerRotationTemplate
- **Source code:** https://github.com/aws-samples/aws-secrets-manager-rotation-lambdas/tree/master/SecretsManagerRotationTemplate/lambda_function.py

Lambda rotation function execution role permissions for AWS Secrets Manager

For the section called "Rotation by Lambda function", when Secrets Manager uses a Lambda function to rotate a secret, Lambda assumes an IAM execution role and provides those credentials to the Lambda function code. For instructions about how to set up automatic rotation, see:

- Automatic rotation for database secrets (console)
- Automatic rotation for non-database secrets (console)
- Automatic rotation (AWS CLI)

The following examples show inline policies for Lambda rotation function execution roles. To create an execution role and attach a permissions policy, see AWS Lambda execution role.

Examples:

- [Policy for a Lambda rotation function execution role]
- [Policy statement for customer managed key]
- [Policy statement for alternating users strategy]

Policy for a Lambda rotation function execution role

The following example policy allows the rotation function to:

- Run Secrets Manager operations for *SecretARN*.
- Create a new password.
- Set up the required configuration if your database or service runs in a VPC. See [Configuring a Lambda function to access resources in a VPC].

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:DescribeSecret",
                "secretsmanager:GetSecretValue",
                "secretsmanager:PutSecretValue",
                "secretsmanager:UpdateSecretVersionStage"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName-AbCdEf"
    },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetRandomPassword"
            ],
            "Resource": "*"
        },
        {
            "Action": [
                "ec2:CreateNetworkInterface",
```

```
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DetachNetworkInterface"
            ],
            "Resource": "*",
            "Effect": "Allow"
        }
    ]
}
```

Policy statement for customer managed key

If the secret is encrypted with a KMS key other than the AWS managed key `aws/secretsmanager`, then you need to grant the Lambda execution role permission to use the key. You can use the SecretARN encryption context to limit the use of the decrypt function, so the rotation function role only has access to decrypt the secret it is responsible for rotating. The following example shows a statement to add to the execution role policy to decrypt the secret using the KMS key.

```
        {
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "KMSKeyARN",
            "Condition": {
                "StringEquals": {
                    "kms:EncryptionContext:SecretARN": "SecretARN"
                }
            }
        }
```

To use the rotation function for multiple secrets that are encrypted with a customer managed key, add a statement like the following example to allow the execution role to decrypt the secret.

```
        {
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
```

```
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "KMSKeyARN",
            "Condition": {
                "StringEquals": {
                    "kms:EncryptionContext:SecretARN": [
                        "arn1",
                        "arn2"
                    ]
                }
            }
        }
```

Policy statement for alternating users strategy

For information about the *alternating users rotation strategy*, see the section called "Lambda function rotation strategies".

For a secret that contains Amazon RDS credentials, if you are using the alternating users strategy and the superuser secret is managed by Amazon RDS, then you must also allow the rotation function to call read-only APIs on Amazon RDS so that it can get the connection information for the database. We recommend you attach the AWS managed policy AmazonRDSReadOnlyAccess.

The following example policy allows the function to:

- Run Secrets Manager operations for *SecretARN*.
- Retrieve the credentials in the superuser secret. Secrets Manager uses the credentials in the superuser secret to update the credentials in the rotated secret.
- Create a new password.
- Set up the required configuration if your database or service runs in a VPC. For more information, see Configuring a Lambda function to access resources in a VPC.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "secretsmanager:DescribeSecret",
                "secretsmanager:GetSecretValue",
                "secretsmanager:PutSecretValue",
                "secretsmanager:UpdateSecretVersionStage"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName-AbCdEf"
    },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName-AbCdEf"
    },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetRandomPassword"
            ],
            "Resource": "*"
        },
        {
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DetachNetworkInterface"
            ],
            "Resource": "*",
            "Effect": "Allow"
        }
    ]
}
```

Network access for AWS Lambda rotation function

For the section called “Rotation by Lambda function”, when Secrets Manager uses a Lambda function to rotate a secret, the Lambda rotation function must be able to access the secret. If your secret contains credentials, then the Lambda function must also be able to access the source of those credentials, such as a database or service.

To access a secret

Your Lambda rotation function must be able to access a Secrets Manager endpoint. If your Lambda function can access the internet, then you can use a public endpoint. To find an endpoint, see the section called “Secrets Manager endpoints”.

If your Lambda function runs in a VPC that doesn't have internet access, we recommend you configure Secrets Manager service private endpoints within your VPC. Your VPC can then intercept requests addressed to the public regional endpoint and redirect them to the private endpoint. For more information, see VPC endpoints (AWS PrivateLink).

Alternatively, you can enable your Lambda function to access a Secrets Manager public endpoint by adding a NAT gateway or an internet gateway to your VPC, which allows traffic from your VPC to reach the public endpoint. This exposes your VPC to more risk because an IP address for the gateway can be attacked from the public Internet.

(Optional) To access the database or service

For secrets such as API keys, there is no source database or service that you need to update along with the secret.

If your database or service is running on an Amazon EC2 instance in a VPC, we recommend that you configure your Lambda function to run in the same VPC. Then the rotation function can communicate directly with your service. For more information, see Configuring VPC access.

To allow the Lambda function to access the database or service, you must make sure that the security groups attached to your Lambda rotation function allow outbound connections to the database or service. You must also make sure that the security groups attached to your database or service allow inbound connections from the Lambda rotation function.

Troubleshoot AWS Secrets Manager rotation

For many services, Secrets Manager uses a Lambda function to rotate secrets. For more information, see the section called “Rotation by Lambda function”. The Lambda rotation function interacts with the database or service the secret is for as well as Secrets Manager. When rotation doesn't work the way you expect, you should first check the CloudWatch logs.

> **Note**
>
> Some services can manage secrets for you, including managing automatic rotation. For more information, see the section called “Managed rotation”.

Topics

How to troubleshoot secret rotation failures in AWS Lambda functions

If you're experiencing secret rotation failures with your Lambda functions, use the following steps to troubleshoot and resolve the issue.

Possible causes

- Insufficient concurrent executions for the Lambda function
- Race conditions due to multiple API calls during rotation
- Incorrect Lambda function logic
- Networking issues between the Lambda function and the database

General troubleshooting steps

1. Analyze CloudWatch logs:

- Look for specific error messages or unexpected behavior in the Lambda function logs
- Verify that all rotation steps (**CreateSecret**, **SetSecret**, **TestSecret**, **FinishSecret**) are being attempted

2. Review API calls during rotation:
   - Avoid making mutating API calls on the secret during Lambda rotation
   - Ensure there's no race condition between **RotateSecret** and **PutSecretValue** calls
3. Verify Lambda function logic:
   - Confirm you're using the latest AWS sample code for secret rotation
   - If using custom code, review it for proper handling of all rotation steps
4. Check network configuration:
   - Verify security group rules allow the Lambda function to access the database
   - Ensure proper VPC endpoint or public endpoint access for Secrets Manager
5. Test secret versions:
   - Verify that the AWSCURRENT version of the secret allows database access
   - Check if AWSPREVIOUS or AWSPENDING versions are valid
6. Clear pending rotations:
   - If rotation consistently fails, clear the AWSPENDING staging label and retry rotation
7. Check Lambda concurrency settings:
   - Verify that concurrency settings are appropriate for your workload
   - If you suspect concurrency issues, see the "Troubleshooting concurrency-related rotation failures" section

No activity after "Found credentials in environment variables"

If there is no activity after "Found credentials in environment variables", and the task duration is long, for example the default Lambda timeout of 30000ms, then the Lambda function may be timing out while trying to reach the Secrets Manager endpoint.

Your Lambda rotation function must be able to access a Secrets Manager endpoint. If your Lambda function can access the internet, then you can use a public endpoint. To find an endpoint, see the section called “Secrets Manager endpoints”.

If your Lambda function runs in a VPC that doesn't have internet access, we recommend you configure Secrets Manager service private endpoints within your VPC. Your VPC can then intercept requests addressed to the public regional endpoint and redirect them to the private endpoint. For more information, see VPC endpoints (AWS PrivateLink).

Alternatively, you can enable your Lambda function to access a Secrets Manager public endpoint by adding a NAT gateway or an internet gateway to your VPC, which allows traffic from your VPC to reach the public endpoint. This exposes your VPC to more risk because an IP address for the gateway can be attacked from the public Internet.

No activity after "createSecret"

The following are issues that can cause rotation to stop after createSecret:

The VPC Network ACLs do not allow HTTPS traffic in and out.

For more information, see Control traffic to subnets using Network ACLs in the *Amazon VPC User Guide*.

Lambda function timeout configuration is too short to perform the task.

For more information, see Configuring Lambda function options in the *AWS Lambda Developer Guide*.

The Secrets Manager VPC endpoint does not allow the VPC CIDRs on ingress in the assigned security groups.

For more information, see Control traffic to resources using security groups in the *Amazon VPC User Guide*.

The Secrets Manager VPC endpoint policy does not allow Lambda to use the VPC endpoint.

For more information, see the section called “VPC endpoints (AWS PrivateLink)”.

The secret uses alternating users rotation, the superuser secret is managed by Amazon RDS, and the Lambda function can't access the RDS API.

For alternating users rotation where the superuser secret is managed by another AWS service, the Lambda rotation function must be able to call the service endpoint to get the database connection information. We recommend that you configure a VPC endpoint for the database service. For more information, see:

- Amazon RDS API and interface VPC endpoints in the *Amazon RDS User Guide*.
- Working with VPC endpoints in the *Amazon Redshift Management Guide*.

Error: "Access to KMS is not allowed"

If you see `ClientError: An error occurred (AccessDeniedException) when calling the GetSecretValue operation: Access to KMS is not allowed`, the rotation function does not have permission to decrypt the secret using the KMS key that was used to encrypt the secret. There might be a condition in the permissions policy that limits the encryption context to a specific secret. For information about the required permission, see the section called "Policy statement for customer managed key".

Error: "Key is missing from secret JSON"

A Lambda rotation function requires the secret value to be in a specific JSON structure. If you see this error, then the JSON might be missing a key that the rotation function tried to access. For information about the JSON structure for each type of secret, see the section called "JSON structure of a secret".

Error: "setSecret: Unable to log into database"

The following are issues that can cause this error:

The rotation function can't access the database.

If the task duration is long, for example over 5000ms, then the Lambda rotation function might not be able to access the database over the network.

If your database or service is running on an Amazon EC2 instance in a VPC, we recommend that you configure your Lambda function to run in the same VPC. Then the rotation function can communicate directly with your service. For more information, see Configuring VPC access.

To allow the Lambda function to access the database or service, you must make sure that the security groups attached to your Lambda rotation function allow outbound connections to the database or service. You must also make sure that the security groups attached to your database or service allow inbound connections from the Lambda rotation function.

The credentials in the secret are incorrect.

If the task duration is short, then the Lambda rotation function might not be able to authenticate with the credentials in the secret. Check the credentials by logging in manually with the information in the `AWSCURRENT` and `AWSPREVIOUS` versions of the secret using the AWS CLI command [`get-secret-value`](get-secret-value).

The database uses `scram-sha-256` to encrypt passwords.

If your database is Aurora PostgreSQL version 13 or later and uses `scram-sha-256` to encrypt passwords, but the rotation function uses `libpq` version 9 or older which does not support `scram-sha-256`, then the rotation function can't connect to the database.

To determine which database users use `scram-sha-256` encryption

- See *Checking for users with non-SCRAM passwords* in the blog SCRAM Authentication in RDS for PostgreSQL 13.

To determine which version of `libpq` your rotation function uses

1. On a Linux-based computer, on the Lambda console, navigate to your rotation function and download the deployment bundle. Uncompress the zip file into a work directory.
2. At a command line, in the work directory, run:

   ```
   readelf -a libpq.so.5 | grep RUNPATH
   ```
3. If you see the string *`PostgreSQL-9.4.x`*, or any major version less than 10, then the rotation function doesn't support `scram-sha-256`.
   - Output for a rotation function that doesn't support `scram-sha-256`:

     ```
     0x000000000000001d (RUNPATH) Library runpath: [/
     local/p4clients/pkgbuild-a1b2c/workspace/build/
     PostgreSQL/PostgreSQL-9.4.x_client_only.123456.0/AL2_x86_64/
     DEV.STD.PTHREAD/build/private/tmp/brazil-path/build.libfarm/lib:/
     local/p4clients/pkgbuild-a1b2c/workspace/src/PostgreSQL/build/
     private/install/lib]
     ```
   - Output for a rotation function that supports `scram-sha-256`:

     ```
     0x000000000000001d (RUNPATH) Library runpath: [/
     local/p4clients/pkgbuild-a1b2c/workspace/build/
     ```

```
PostgreSQL/PostgreSQL-10.x_client_only.123456.0/AL2_x86_64/
DEV.STD.PTHREAD/build/private/tmp/brazil-path/build.libfarm/lib:/
local/p4clients/pkgbuild-a1b2c/workspace/src/PostgreSQL/build/
private/install/lib]
```

- Output for a rotation function that supports `scram-sha-256`:

```
0x000000000000001d (RUNPATH) Library runpath: [/local/
p4clients/pkgbuild- a1b2c /workspace/build/PostgreSQL/
PostgreSQL-14.x_client_only. 123456 .0/AL2_x86_64/
DEV.STD.PTHREAD/build/private/tmp/brazil-path/build.libfarm/lib:/
local/p4clients/pkgbuild- a1b2c /workspace/src/PostgreSQL/build/
private/install/lib]
```

- Output for a rotation function that supports `scram-sha-256`:

```
0x000000000000001d (RUNPATH) Library runpath: [/local/p4clients/
pkgbuild- a1b2c/workspace/build/PostgreSQL/PostgreSQL-
14.x_client_only.123456.0/AL2_x86_64/DEV.STD.PTHREAD/build/
private/tmp/brazil- path/build.libfarm/lib:/local/p4clients/
pkgbuild- a1b2c/workspace/src/PostgreSQL/build/private/install/
lib]
```

> **Note**
>
> If you set up automatic secret rotation before December 30, 2021, your rotation function bundled an earlier version of `libpq` that doesn't support `scram-sha-256`. To support `scram-sha-256`, you need to recreate your rotation function.

The database requires SSL/TLS access.

If your database requires an SSL/TLS connection, but the rotation function uses an unencrypted connection, then the rotation function can't connect to the database. Rotation functions for Amazon RDS (except Oracle and Db2) and Amazon DocumentDB automatically use Secure Socket Layer (SSL) or Transport Layer Security (TLS) to connect to your database, if it is available. Otherwise they use an unencrypted connection.

> **Note**
>
> If you set up automatic secret rotation before December 20, 2021, your rotation function might be based on an earlier template that did not support SSL/TLS. To support connections that use SSL/TLS, you need to recreate your rotation function.

To determine when your rotation function was created

1. In the Secrets Manager console https://console.aws.amazon.com/secretsmanager/, open your secret. In the **Rotation configuration** section, under **Lambda rotation function**, you see the **Lambda function ARN**, for example, `arn:aws:lambda:`*`aws-region`*`:`*`123456789012`*`:function:`*`SecretsManagerMyRotationFunction`* . Copy the function name from the end of the ARN, in this example *`SecretsManagerMyRotationFunction`* .
2. In the AWS Lambda console https://console.aws.amazon.com/lambda/, under **Functions**, paste your Lambda function name in the search box, choose Enter, and then choose the Lambda function.
3. In the function details page, on the **Configuration** tab, under **Tags**, copy the value next to the key **aws:cloudformation:stack-name**.
4. In the AWS CloudFormation console https://console.aws.amazon.com/cloudformation, under **Stacks**, paste the key value in the search box, and then choose Enter.
5. The list of stacks filters so that only the stack that created the Lambda rotation function appears. In the **Created date** column, view the date the stack was created. This is the date the Lambda rotation function was created.

Error: "Unable to import module 'lambda_function'"

You might receive this error if you're running an earlier Lambda function that was automatically upgraded from Python 3.7 to a newer version of Python. To resolve the error, you can change the Lambda function version back to Python 3.7, and then the section called "Upgrade an existing rotation function from Python 3.7 to 3.9". For more information, see Why did my Secrets Manager Lambda function rotation fail with a "pg module not found" error? in *AWS re:Post*.

Upgrade an existing rotation function from Python 3.7 to 3.9

Some rotation functions created before November 2022 used Python 3.7. The AWS SDK for Python stopped supporting Python 3.7 in December 2023. For more information, see Python support policy updates for AWS SDKs and Tools. To switch to a new rotation function that uses Python 3.9, you can add a runtime property to an existing rotation function or recreate the rotation function.

To find which Lambda rotation functions use Python 3.7

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. In the list of **Functions**, filter for **`SecretsManager`**.
3. In the filtered list of functions, under **Runtime**, look for Python 3.7.

To upgrade to Python 3.9:

- Option 1: Recreate the rotation function using CloudFormation
- Option 2: Update the runtime for the existing rotation function using CloudFormation
- Option 3: For AWS CDK users, upgrade the CDK library

Option 1: Recreate the rotation function using CloudFormation

When you use the Secrets Manager console to turn on rotation, Secrets Manager uses CloudFormation to create the necessary resources, including the Lambda rotation function. If you used the console to turn on rotation, or you created the rotation function using a CloudFormation stack, you can use the same CloudFormation stack to recreate the rotation function with a new name. The new function uses the more recent version of Python.

To find the CloudFormation stack that created the rotation function

- On the Lambda function details page, on the **Configuration** tab, choose **Tags**. View the ARN next to **aws:cloudformation:stack-id**.

  The stack name is embedded in the ARN, as shown in the following example.

  - ARN: `arn:aws:cloudformation:us-west-2:408736277230:stack/`**`SecretsManagerRDSMySQLRotationSingleUser5c2-SecretRotationScheduleHostedRotationLambda`**`-3CUDHZMDMBO8/79fc9050-2eef-11ed-`

- Stack name: **`SecretsManagerRDSMySQLRotationSingleUser5c2-SecretRotationScheduleHostedRotationLambda`**

To recreate a rotation function (CloudFormation)

1. In CloudFormation, search for the stack by name, and then choose **Update**.

   If a dialog box appears recommending you update the root stack, choose **Go to root stack**, and then choose **Update**.
2. On the **Update stack** page, under **Prepare template**, choose **Edit in Application Composer**, and then under **Edit template in Application Composer**, choose the button **Edit in Application Composer**.
3. In Application Composer, do the following:

   a. In the template code, in `SecretRotationScheduleHostedRotationLambda`, replace the value for `"functionName": "SecretsManagerTestRotationRDS"` with a new function name, for example in JSON, **`"functionName": "SecretsManagerTestRotationRDSupdated"`**

   b. Choose **Update template**.

   c. In the **Continue to CloudFormation** dialog box, choose **Confirm and continue to CloudFormation**.
4. Continue through the CloudFormation stack workflow and then choose **Submit**.

Option 2: Update the runtime for the existing rotation function using CloudFormation

When you use the Secrets Manager console to turn on rotation, Secrets Manager uses CloudFormation to create the necessary resources, including the Lambda rotation function. If you used the console to turn on rotation, or you created the rotation function using a CloudFormation stack, you can use the same CloudFormation stack to update the runtime for the rotation function.

To find the CloudFormation stack that created the rotation function

- On the Lambda function details page, on the **Configuration** tab, choose **Tags**. View the ARN next to **aws:cloudformation:stack-id**.

  The stack name is embedded in the ARN, as shown in the following example.

- ARN: `arn:aws:cloudformation:us-west-2:408736277230:stack/`**`SecretsManagerRDSMySQLRotationSingleUser5c2-SecretRotationScheduleHostedRotationLambda`**`-3CUDHZMDMB08/79fc9050-2eef-11ed-`
- Stack name: **`SecretsManagerRDSMySQLRotationSingleUser5c2-SecretRotationScheduleHostedRotationLambda`**

To update the runtime for a rotation function (CloudFormation)

1. In CloudFormation, search for the stack by name, and then choose **Update**.

   If a dialog box appears recommending you update the root stack, choose **Go to root stack**, and then choose **Update**.
2. On the **Update stack** page, under **Prepare template**, choose **Edit in Application Composer**, and then under **Edit template in Application Composer**, choose the button **Edit in Application Composer**.
3. In Application Composer, do the following:

   a. In the template JSON, for the `SecretRotationScheduleHostedRotationLambda`, under `Properties`, under `Parameters`, add **`"runtime": "python3.9"`**.

   b. Choose **Update template**.

   c. In the **Continue to CloudFormation** dialog box, choose **Confirm and continue to CloudFormation**.
4. Continue through the CloudFormation stack workflow and then choose **Submit**.

Option 3: For AWS CDK users, upgrade the CDK library

If you used the AWS CDK prior to version v2.94.0 to set up rotation for your secret, you can update the Lambda function by upgrading to v2.94.0 or later. For more information, see the AWS Cloud Development Kit (AWS CDK) v2 Developer Guide.

Upgrade an existing rotation function from Python 3.9 to 3.10

Secrets Manager is transitioning from Python 3.9 to 3.10 for Lambda rotation functions. To switch to a new rotation function that uses Python 3.10, you'll need to follow the upgrade path based on your deployment method. Use the following procedures to upgrade both the Python version and the underlying dependencies.

To find which Lambda rotation functions use Python 3.9

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. In the list of **Functions**, filter for **`SecretsManager`**.
3. In the filtered list of functions, under **Runtime**, look for **`Python 3.9`**.

Update paths by deployment method

The Lambda rotation functions identified in this list can be deployed through Secrets Manager console, AWS Serverless Application Repository apps, or CloudFormation transforms. Each of these deployment strategies have a distinct update path.

Use one of the following procedures to update your Lambda rotation functions, depending on how your function was deployed.

AWS Secrets Manager console-deployed functions

A new Lambda function must be deployed through AWS Secrets Manager console as you cannot manually update dependencies for existing Lambda functions.

Use the following procedure to upgrade AWS Secrets Manager console-deployed functions.

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Under **AWS Secrets Manager**, select **Secrets**. Select the secret that uses the Lambda function you want to update.
3. Navigate to the **Rotations** tab and select the **Update rotation configurations** option.
4. Under **Rotation functions**, choose **Create a new function**, and enter a new name for the Lambda rotation function.

    a. (Optional) Once the update is complete, you can test the updated Lambda function to confirm it works as expected. Under the **Rotation** tab, select **Rotate Secret Immediately** to initiate an immediate rotation.

    b. (Optional) You can view your function logs and the Python version used at runtime in Amazon CloudWatch. For more information, see [Viewing CloudWatch Logs for Lambda functions]() in the *AWS Lambda Developer Guide*.
5. Once the new rotation function is set up, you can delete the old rotation function.

AWS Serverless Application Repository deployments

The following procedure shows how to upgrade AWS Serverless Application Repository deployments. The Lambda functions deployed through AWS Serverless Application Repository have a banner stating `This function belongs to an application. Click here to manage it.` which includes a link to the Lambda application to which the function belongs.

> ⚠ **Important**
>
> AWS Serverless Application Repository availability is AWS Region dependent.

Use the following procedure to update AWS Serverless Application Repository deployed functions.

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Navigate to the **Configurations** tab of the Lambda function that needs to be updated.
    - You'll need the following information about your function when updating the deployed AWS Serverless Application Repository application. You can find this information in the Lambda console.
        - **Lambda application's name**
            - The Lambda application name can be found by using the link in the banner. For example, the banner states the following `serverlessrepo-`*`SecretsManagerRedshiftRotationSingleUser`*. The name in this example is `SecretsManagerRedshiftRotationSingleUser`.
        - **Lambda rotation function name**
        - **Secrets Manager endpoint**
            - The endpoint can be found under the **Configurations** and the **Environment variables** tabs assigned to the **SECRETS_MANAGER_ENDPOINT** variable.
3. To upgrade Python, you must update the semantic version of the serverless application. See [Updating Applications](#) in the *AWS Serverless Application Repository Developer Guide*.

Custom Lambda rotation functions

If you created custom Lambda rotation functions, you'll need to upgrade each package dependencies and runtimes for these functions. For more information, see Upgrade Lambda function runtime to latest version.

AWS::SecretsManager-2024-09-16 transform macro

If the Lambda function is deployed through this transform, updating the stacks using existing template will allow you to use the updated Lambda runtime.

Use the following procedure to update CloudFormation stack using existing template.

1. Open the CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. On the **Stacks** page, select the stack that you want to update.
3. Choose **Update** on the stack details pane.
4. For **Choose a template update method**, select **Direct update**.
5. On the **Specify template** page, select **Use existing template**.
6. Keep all other options at their default values, and then choose **Update stack**.

If you experience issues updating the stack, see Determine the cause of a stack failure in the *CloudFormation User Guide*.

AWS::SecretsManager-2020-07-23 transform macro

We recommend you migrate to the newer transform version if you're using `AWS::SecretsManager-2020-07-23`. See Introducing an enhanced version of the AWS Secrets Manager transform: AWS::SecretsManager-2024-09-16 in the *AWS Security Blog* for more information. If you continue to use `AWS::SecretsManager-2020-07-23`, you can experience a mismatch error between your runtime version and the Lambda function code artifacts. For more information, see AWS::SecretsManager::RotationSchedule HostedRotationLambda in the *CloudFormation Template Reference*.

If you experience issues updating the stack, Determine the cause of a stack failure in the *CloudFormation User Guide*.

Verify Python upgrade

To verify the Python upgrade, open the Lambda console (https://console.aws.amazon.com/lambda/) and access the **Function** page. Select the function you updated. Under **Code source** section, review the files included in the directory and ensure the Python .so file is version `3.10`.

AWS Lambda secret rotation with `PutSecretValue` failed

If you use an assumed role or a cross-account rotation with Secrets Manager and you find a **RotationFailed** event in CloudTrail with the message: Pending secret version *`VERSION_ID`* for Secret *`SECRET_ARN`* was not created by Lambda *`LAMBDA_ARN`*. Remove the `AWSPENDING` staging label and restart rotation, then you need to update your Lambda function to use the `RotationToken` parameter.

Update Lambda rotation function to include `RotationToken`

1. Download the Lambda function code
   - Open the Lambda console
   - In the navigation pane, choose **Functions**
   - Select your Lambda secret rotation function for **Function name**
   - For **Download**, choose one of **Function code .zip**, **AWS SAM file**, **Both**
   - Choose **OK** to save the function on your local machine.
2. Edit `Lambda_handler`

   Include the rotation_token parameter in the create_secret step for cross-account rotation:

```
def lambda_handler(event, context):
    """Secrets Manager Rotation Template

    This is a template for creating an AWS Secrets Manager rotation lambda

    Args:
        event (dict): Lambda dictionary of event parameters. These keys must
 include the following:
            - SecretId: The secret ARN or identifier
            - ClientRequestToken: The ClientRequestToken of the secret version
            - Step: The rotation step (one of createSecret, setSecret, testSecret,
 or finishSecret)
            - RotationToken: the rotation token to put as parameter for
 PutSecretValue call
```

```
        context (LambdaContext): The Lambda runtime information

    Raises:
        ResourceNotFoundException: If the secret with the specified arn and stage
does not exist

        ValueError: If the secret is not properly configured for rotation

        KeyError: If the event parameters do not contain the expected keys

    """
    arn = event['SecretId']
    token = event['ClientRequestToken']
    step = event['Step']
    # Add the rotation token
    rotation_token = event['RotationToken']

    # Setup the client
    service_client = boto3.client('secretsmanager',
endpoint_url=os.environ['SECRETS_MANAGER_ENDPOINT'])

    # Make sure the version is staged correctly
    metadata = service_client.describe_secret(SecretId=arn)
    if not metadata['RotationEnabled']:
        logger.error("Secret %s is not enabled for rotation" % arn)
        raise ValueError("Secret %s is not enabled for rotation" % arn)
    versions = metadata['VersionIdsToStages']
    if token not in versions:
        logger.error("Secret version %s has no stage for rotation of secret %s." %
(token, arn))
        raise ValueError("Secret version %s has no stage for rotation of secret
%s." % (token, arn))
    if "AWSCURRENT" in versions[token]:
        logger.info("Secret version %s already set as AWSCURRENT for secret %s." %
(token, arn))
        return
    elif "AWSPENDING" not in versions[token]:
        logger.error("Secret version %s not set as AWSPENDING for rotation of
secret %s." % (token, arn))
        raise ValueError("Secret version %s not set as AWSPENDING for rotation of
secret %s." % (token, arn))
    # Use rotation_token
    if step == "createSecret":
```

```
        create_secret(service_client, arn, token, rotation_token)

    elif step == "setSecret":
        set_secret(service_client, arn, token)

    elif step == "testSecret":
        test_secret(service_client, arn, token)

    elif step == "finishSecret":
        finish_secret(service_client, arn, token)

    else:
        raise ValueError("Invalid step parameter")
```

3. Edit `create_secret` code

   Revise the `create_secret` function to accept and use the `rotation_token` parameter:

```
# Add rotation_token to the function
def create_secret(service_client, arn, token, rotation_token):
"""Create the secret

This method first checks for the existence of a secret for the passed in token. If
 one does not exist, it will generate a
new secret and put it with the passed in token.

Args:
service_client (client): The secrets manager service client

arn (string): The secret ARN or other identifier

token (string): The ClientRequestToken associated with the secret version

rotation_token (string): the rotation token to put as parameter for PutSecretValue
 call

Raises:
ResourceNotFoundException: If the secret with the specified arn and stage does not
 exist

"""
# Make sure the current secret exists
```

```
service_client.get_secret_value(SecretId=arn, VersionStage="AWSCURRENT")

# Now try to get the secret version, if that fails, put a new secret
try:
service_client.get_secret_value(SecretId=arn, VersionId=token,
 VersionStage="AWSPENDING")
logger.info("createSecret: Successfully retrieved secret for %s." % arn)
except service_client.exceptions.ResourceNotFoundException:
# Get exclude characters from environment variable
exclude_characters = os.environ['EXCLUDE_CHARACTERS'] if 'EXCLUDE_CHARACTERS' in
 os.environ else '/@"\'\\'
# Generate a random password
passwd = service_client.get_random_password(ExcludeCharacters=exclude_characters)

# Put the secret, using rotation_token
service_client.put_secret_value(SecretId=arn, ClientRequestToken=token,
 SecretString=passwd['RandomPassword'], VersionStages=['AWSPENDING'],
 RotationToken=rotation_token)
logger.info("createSecret: Successfully put secret for ARN %s and version %s." %
 (arn, token))
```

4. Upload the updated Lambda function code

   After updating your Lambda function code, upload it to rotate your secret.

Error: "Error when executing lambda *<arn>* during *<a rotation>* step"

If you're experiencing intermittent secret rotation failures with your Lambda function getting stuck in a loop of sets, for example between **CreateSecret** and **SetSecret**, the issue may be related to concurrency settings.

Concurrency troubleshooting steps

> **Warning**
>
> Setting the provisioned concurrency parameter to a value lower than 10 can cause throttling due to insufficient execution threads for the Lambda function. For more information, see Understanding reserved concurrency and provisioned concurrency in the AWS Lambda AWS Lambda Developer Guide.

1. Check and adjust Lambda concurrency settings:
   - Verify that `reserved_concurrent_executions` is not set too low (for example, 1)
   - If using reserved concurrency, set it to at least 10
   - Consider using unreserved concurrency for more flexibility
2. For provisioned concurrency:
   - Don't set the provisioned concurrency parameter explicitly (for example, in Terraform).
   - If you must set it, use a value of at least 10.
   - Test thoroughly to make sure the chosen value works for your use case.
3. Monitor and adjust concurrency:
   - Calculate concurrency using this formula: Concurrency = (average requests per second) * (average request duration in seconds). For more information, see Estimating reserved concurrency.
   - Observe and record values during rotations to determine the appropriate concurrency settings.
   - Be careful when setting low concurrency values. They can cause throttling if there aren't enough available execution threads.

For more information on configuring Lambda concurrency, see Configuring reserved concurrency and Configuring provisioned concurrency in the AWS Lambda Developer Guide.

Rotation schedules

Secrets Manager rotates your secret on a schedule during a rotation window that you set. To set the schedule and window, you use a **cron()** or **rate()** expression along with a window duration. Secrets Manager rotates your secret at any time during the rotation window. You can rotate a secret as often as every four hours within a rotation window as small as one hour.

To turn on rotation, see:

- the section called “Managed rotation”
- the section called “Automatic rotation for database secrets (console)”
- the section called “Automatic rotation for non-database secrets (console)”

Secrets Manager rotation schedules use UTC time zone.

Rotation windows

A Secrets Manager rotation window is similar to a maintenance window. You set the rotation window when you want your secret rotated, and Secrets Manager rotates your secret at some time during the rotation window.

Secrets Manager rotation windows always start on the hour. For a rotation schedule that uses a `rate()` expression in days, the rotation window starts at midnight. You can set the start time for the rotation window by using a `cron()` expression. For examples, see the section called "Cron expressions".

By default, the rotation window closes after one hour for a rotation schedule in *hours*, and at the end of the day for a rotation schedule in *days*.

To change the length of the rotation window, set the **Window duration**. You can set the rotation window as small as one hour. The rotation window must not extend into the next rotation window. In other words, for a rotation schedule in *hours*, confirm that the rotation window is less than or equal to the number of hours between rotations. For a rotation schedule in *days*, confirm that the start hour plus the window duration is less than or equal to 24 hours.

Rate expressions

Secrets Manager rate expressions have the following format, where *`Value`* is a positive integer and *`Unit`* can be `hour`, `hours`, `day`, or `days`:

```
rate(Value Unit)
```

You can rotate a secret as often as every four hours. The maximum rotation period is 999 days. Examples:

- `rate(4 hours)` means the secret is rotated every four hours.
- `rate(1 day)` means the secret is rotated every day.
- `rate(10 days)` means the secret is rotated every 10 days.

Cron expressions

Secrets Manager cron expressions have the following format:

```
cron(Minutes Hours Day-of-month Month Day-of-week Year)
```

A cron expression that includes increments of hours resets each day. For example, `cron(0 4/12 * * ? *)` means 4:00 AM, 4:00 PM, and then the next day 4:00 AM, 4:00 PM. Secrets Manager rotation schedules use UTC time zone.

Example schedule	Expression
Every eight hours starting at midnight.	`cron(0 /8 * * ? *)`
Every eight hours starting at 8:00 AM.	`cron(0 8/8 * * ? *)`
Every ten hours, starting at 2:00 AM. The rotation windows will start at 2:00, 12:00, and 22:00, and then the next day at 2:00, 12:00, and 22:00.	`cron(0 2/10 * * ? *)`
Every day at 10:00 AM.	`cron(0 10 * * ? *)`
Every Saturday at 6:00 PM.	`cron(0 18 ? * SAT *)`
The first day of every month at 8:00 AM.	`cron(0 8 1 * ? *)`
Every three months on the first Sunday at 1:00 AM.	`cron(0 1 ? 1/3 SUN#1 *)`
The last day of every month at 5:00 PM.	`cron(0 17 L * ? *)`
Monday through Friday at 8:00 AM.	`cron(0 8 ? * MON-FRI *)`
First and 15th day of every month at 4:00 PM.	`cron(0 16 1,15 * ? *)`
First Sunday of every month at midnight.	`cron(0 0 ? * SUN#1 *)`
Starting in January, every 11 months on the first Monday at midnight.	`cron(0 0 ? 1/11 2#1 *)`

Cron expression requirements in Secrets Manager

Secrets Manager has some restrictions on what you can use for cron expressions. A cron expression for Secrets Manager must have **0** in the minutes field because Secrets Manager rotation windows start on the hour. It must have ***** in the year field, because Secrets Manager does not support rotation schedules that are more than a year apart. The following table shows the options you can use.

Fields	Values	Wildcards
Minutes	Must be 0	None
Hours	0–23	Use **/** (forward slash) to specify increments. For example `2/10` means every 10 hours beginning at 2:00 AM. You can rotate a secret as often as every four hours.
Day-of-month	1–31	Use **,** (comma) to include additional values. For example `1,15` means the first and 15th day of the month. Use **-** (dash) to specify a range. For example 1–15 means days 1 through 15 of the month. Use ***** (asterisk) to includes all values in the field. For example `*` means every day of the month. The **?** (question mark) wildcard specifies one or another. You can't specify the `Day-of-month` and `Day-`

Fields	Values	Wildcards
		`of-week` fields in the same cron expression. If you specify a value in one of the fields, you must use a **?** (question mark) in the other. Use **/** (forward slash) to specify increments. For example, `1/2` means every two days starting on day 1, in other words, day 1, 3, 5, and so on. Use **L** to specify the last day of the month. Use ***DAY*L** to specify the last named day of the month. For example `SUNL` means the last Sunday of the month.

Fields	Values	Wildcards
Month	1–12 or JAN–DEC	Use **,** (comma) to include additional values. For example, `JAN,APR,JUL,OCT` means January, April, July, and October. Use **-** (dash) to specify a range. For example 1–3 means months 1 through 3 of the year. Use ***** (asterisk) to includes all values in the field. For example `*` means every month. Use **/** (forward slash) to specify increments. For example, `1/3` means every third month, starting on month 1, in other words month 1, 4, 7, and 10.

Fields	Values	Wildcards
Day-of-week	1–7 or SUN–SAT	Use **#** to specify the day of the week within a month. For example, `TUE#3` means the third Tuesday of the month. Use **,** (comma) to include additional values. For example `1,4` means the first and fourth day of the week. Use **-** (dash) to specify a range. For example `1–4` means days 1 through 4 of the week. Use ***** (asterisk) to includes all values in the field. For example `*` means every day of the week. The **?** (question mark) wildcard specifies one or another. You can't specify the `Day-of-month` and `Day-of-week` fields in the same cron expression. If you specify a value in one of the fields, you must use a **?** (question mark) in the other. Use **/** (forward slash) to specify increments. For example, `1/2` means every second day of the week, starting on the first day, so day 1, 3, 5, and 7.

Fields	Values	Wildcards
		Use **L** to specify the last day of the week.
Year	Must be *	None

Rotate an AWS Secrets Manager secret immediately

You can only rotate a secret that has rotation configured. To determine whether a secret has been configured for rotation, in the console, view the secret and scroll down to the **Rotation configuration** section. If **Rotation status** is **Enabled**, then the secret is configured for rotation. If not, see *Rotate secrets*.

To rotate a secret immediately (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose your secret.
3. On the secret details page, under **Rotation configuration**, choose **Rotate secret immediately**.
4. In the **Rotate secret** dialog box, choose **Rotate**.

AWS CLI

Example Rotate a secret immediately

The following **rotate-secret** example starts an immediate rotation. The secret must already have rotation configured.

```
$ aws secretsmanager rotate-secret \
    --secret-id MyTestSecret
```

Find secrets that aren't rotated

You can use AWS Config to evaluate your secrets to see if they are rotating in compliance with your standards. You define your internal security and compliance requirements for secrets using AWS Config rules. Then AWS Config can identify secrets that don't conform to your rules. You can also track changes to secret metadata, rotation configuration, the KMS key used for secret encryption, the Lambda rotation function, and tags associated with a secret.

If you have secrets in multiple AWS accounts and AWS Regions in your organization, you can aggregate that configuration and compliance data. For more information, see Multi-account Multi-Region data aggregation.

To assess whether secrets are rotating

1. Follow the instructions on Evaluating your resources with AWS Config rules, and choose from of the following rules:
   - `secretsmanager-rotation-enabled-check` — Checks whether rotation is configured for secrets stored in Secrets Manager.
   - `secretsmanager-scheduled-rotation-success-check`— Checks whether the last successful rotation is within the configured rotation frequency. The minimum frequency for the check is daily.
   - `secretsmanager-secret-periodic-rotation`— Checks whether secrets were rotated within the specified number of days.
2. Optionally, configure AWS Config to notify you when secrets aren't compliant. For more information, see Notifications that AWS Config sends to an Amazon SNS topic.

Cancel automatic rotation in Secrets Manager

If you configured automatic rotation for a secret and you want to stop rotating it, you can cancel rotation.

To cancel automatic rotation

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. Choose your secret.
3. On the secret details page, under **Rotation configuration**, choose **Edit rotation**.
4. In the **Edit rotation configuration** dialog box, turn off **Automatic rotation**, and then choose **Save**.

   Secrets Manager retains the rotation configuration information so that you can use it in the future if you decide to turn rotation back on.

AWS Secrets Manager secrets managed by other AWS services

Many AWS services store and use secrets in AWS Secrets Manager. In some cases, these secrets are *managed secrets*, which means that the service that created them helps manage them. For example, some managed secrets include managed rotation, so you don't have to configure rotation yourself. The managing service might also restrict you from updating secrets or deleting them without a recovery period, which helps prevent outages because the managing service depends on the secret.

> **Note**
>
> Managed secrets can only be created by the AWS service that manages them.

Managed secrets use a naming convention that includes the managing service ID to help identify them.

```
Secret name: ServiceID!MySecret
Secret ARN : arn:aws:us-east-1:ServiceID!MySecret-a1b2c3
```

IDs for services that manage secrets

- `appflow` – the section called “Amazon AppFlow”
- `databrew` – the section called “AWS Glue DataBrew”
- `datasync` – the section called “AWS DataSync”
- `directconnect` – the section called “Direct Connect”
- `ecs-sc` – the section called “Amazon Elastic Container Service”
- `events` – the section called “Amazon EventBridge”
- `marketplace-deployment` – the section called “AWS Marketplace”
- `opsworks-cm` – the section called “AWS OpsWorks for Chef Automate”
- `pcs` – the section called “AWS Parallel Computing Service”
- `rds` – the section called “Amazon RDS”

- `redshift` – the section called “Amazon Redshift”
- `sqlworkbench` – the section called “Amazon Redshift query editor v2”

To find secrets that are managed by other AWS services, see Find managed secrets.

AWS services that use AWS Secrets Manager secrets

Get information about how each of the following AWS services integrate with Secrets Manager.

- How AWS App Runner uses AWS Secrets Manager
- How AWS App2Container uses AWS Secrets Manager
- How AWS AppConfig uses AWS Secrets Manager
- How Amazon AppFlow uses AWS Secrets Manager
- How AWS AppSync uses AWS Secrets Manager
- How Amazon Athena uses AWS Secrets Manager
- How Amazon Aurora uses AWS Secrets Manager
- How AWS CodeBuild uses AWS Secrets Manager
- How Amazon Data Firehose uses AWS Secrets Manager
- How AWS DataSync uses AWS Secrets Manager
- How Amazon DataZone uses AWS Secrets Manager
- How AWS Direct Connect uses AWS Secrets Manager
- How AWS Directory Service uses AWS Secrets Manager
- How Amazon DocumentDB (with MongoDB compatibility) uses AWS Secrets Manager
- How AWS Elastic Beanstalk uses AWS Secrets Manager
- How Amazon Elastic Container Registry uses AWS Secrets Manager
- Amazon Elastic Container Service
- How Amazon ElastiCache uses AWS Secrets Manager
- How AWS Elemental Live uses AWS Secrets Manager
- How AWS Elemental MediaConnect uses AWS Secrets Manager
- How AWS Elemental MediaConvert uses AWS Secrets Manager
- How AWS Elemental MediaLive uses AWS Secrets Manager

- How AWS Elemental MediaPackage uses AWS Secrets Manager
- How AWS Elemental MediaTailor uses AWS Secrets Manager
- How Amazon EMR uses Secrets Manager
- How Amazon EventBridge uses AWS Secrets Manager
- How Amazon FSx uses AWS Secrets Manager secrets
- How AWS Glue DataBrew uses AWS Secrets Manager
- How AWS Glue Studio uses AWS Secrets Manager
- How AWS IoT SiteWise uses AWS Secrets Manager
- How Amazon Kendra uses AWS Secrets Manager
- How Amazon Kinesis Video Streams uses AWS Secrets Manager
- How AWS Launch Wizard uses AWS Secrets Manager
- How Amazon Lookout for Metrics uses AWS Secrets Manager
- How Amazon Managed Grafana uses AWS Secrets Manager
- How AWS Managed Services uses AWS Secrets Manager
- How Amazon Managed Streaming for Apache Kafka uses AWS Secrets Manager
- How Amazon Managed Workflows for Apache Airflow uses AWS Secrets Manager
- AWS Marketplace
- How AWS Migration Hub uses AWS Secrets Manager
- How AWS Panorama uses Secrets Manager
- How AWS Parallel Computing Service uses AWS Secrets Manager
- How AWS ParallelCluster uses AWS Secrets Manager
- How Amazon Q uses Secrets Manager
- How Amazon OpenSearch Ingestion uses Secrets Manager
- How AWS OpsWorks for Chef Automate uses AWS Secrets Manager
- How Amazon Quick Suite uses AWS Secrets Manager
- How Amazon RDS uses AWS Secrets Manager
- How Amazon Redshift uses AWS Secrets Manager
- Amazon Redshift query editor v2
- How Amazon SageMaker AI uses AWS Secrets Manager

- How AWS Schema Conversion Tool uses AWS Secrets Manager
- How Amazon Timestream for InfluxDB uses AWS Secrets Manager
- How AWS Toolkit for JetBrains uses AWS Secrets Manager
- How AWS Transfer Family uses AWS Secrets Manager secrets
- How AWS Wickruses AWS Secrets Manager secrets

How AWS App Runner uses AWS Secrets Manager

AWS App Runner is an AWS service that provides a fast, simple, and cost-effective way to deploy from source code or a container image directly to a scalable and secure web application in the AWS Cloud. You don't need to learn new technologies, decide which compute service to use, or know how to provision and configure AWS resources.

With App Runner, you can reference secrets and configurations as environment variables in your service when you create a service or update the service's configuration. For more information, see Referencing environment variables and Managing environment variables in the *AWS App Runner Developer Guide*.

How AWS App2Container uses AWS Secrets Manager

AWS App2Container is a command line tool to help you lift and shift applications that run in your on-premises data centers or on virtual machines, so that they run in containers that are managed by Amazon ECS, Amazon EKS, or AWS App Runner.

App2Container uses Secrets Manager to manage the credentials for connecting your worker machine to application servers in order to run remote commands. For more information, see Manage secrets for AWS App2Container in the *AWS App2Container User Guide*.

How AWS AppConfig uses AWS Secrets Manager

AWS AppConfig is a capability of AWS Systems Manager that you can use to create, manage, and quickly deploy application configurations. A configuration can contain credential data or other sensitive information stored in Secrets Manager. When you create a freeform configuration profile, you can choose Secrets Manager as the source of your configuration data. For more information, see Creating a freeform configuration profile in the *AWS AppConfig User Guide*. For information about how AWS AppConfig handles secrets that have automatic rotation turned on, see Secrets Manager key rotation in the *AWS AppConfig User Guide*.

How Amazon AppFlow uses AWS Secrets Manager

Amazon AppFlow is a fully-managed integration service that enables you to securely exchange data between software as a service (SaaS) applications, such as Salesforce, and AWS services, such as Amazon Simple Storage Service (Amazon S3) and Amazon Redshift.

In Amazon AppFlow, when you configure an SaaS application as a source or destination, you create a connection. This includes information required for connecting to the SaaS applications, such as authentication tokens, user names, and passwords. Amazon AppFlow stores your connection data in a Secrets Manager managed secret with the prefix `appflow`. The cost of storing the secret is included with the charge for Amazon AppFlow. For more information, see Data protection in Amazon AppFlow in the *Amazon AppFlow User Guide*.

How AWS AppSync uses AWS Secrets Manager

AWS AppSync provides a robust, scalable GraphQL interface for application developers to combine data from multiple sources, including Amazon DynamoDB, AWS Lambda, and HTTP APIs.

AWS AppSync uses the credentials in a Secrets Manager secret to connect to Amazon RDS and Aurora. For more information, see Tutorial: Aurora Serverless in the *AWS AppSync Developer Guide*.

How Amazon Athena uses AWS Secrets Manager

Amazon Athena is an interactive query service that makes it easy to analyze data directly in Amazon Simple Storage Service (Amazon S3) using standard SQL.

Amazon Athena data source connectors can use the Athena Federated Query feature with Secrets Manager secrets to query data. For more information, see Using Amazon Athena Federated Query in the *Amazon Athena User Guide*.

How Amazon Aurora uses AWS Secrets Manager

Amazon Aurora is a fully managed relational database engine that's compatible with MySQL and PostgreSQL.

To manage master user credentials for Aurora, Aurora can create a managed secret for you. You are charged for that secret. Aurora also manages rotation for these credentials. For more information, see Password management with Amazon Aurora and AWS Secrets Manager in the *Amazon Aurora User Guide*.

For other Aurora credentials, see *Create secrets*.

When you call the Amazon RDS Data API, you can pass credentials for the database by using a secret in Secrets Manager. For more information, see Using the Data API for Aurora Serverless in the *Amazon Aurora User Guide*.

When you use the Amazon RDS query editor to connect to a database, you can store credentials for the database in Secrets Manager. For more information, see Using the query editor in the *Amazon RDS User Guide*.

How AWS CodeBuild uses AWS Secrets Manager

AWS CodeBuild is a fully managed build service in the cloud. CodeBuild compiles your source code, runs unit tests, and produces artifacts ready to deploy.

You can store your private registry credentials using Secrets Manager. For more information, see Private registry with AWS Secrets Manager sample for CodeBuild in the *AWS CodeBuild User Guide*.

How Amazon Data Firehose uses AWS Secrets Manager

You can use Amazon Data Firehose to deliver real-time streaming data to various streaming destinations. When the destination requires a credentials or key, Firehose retrieves a secret from Secrets Manager at runtime to connect to the destination. For more information, see Authenticate with AWS Secrets Manager in Amazon Data Firehose in the *Amazon Data Firehose Developer Guide*.

How AWS DataSync uses AWS Secrets Manager

AWS DataSync is an online data transfer service that simplifies, automates, and accelerates moving data between storage systems and services.

Some of the storage systems supported by DataSync require credentials to read and write data. DataSync uses Secrets Manager to store or access storage credentials. You can configure DataSync to create secrets on your behalf or you can provide a custom secret. Service-managed secrets begin with the prefix `aws-datasync`. You are charged only for the use of secrets that you create outside of DataSync. See Providing credentials for storage locations in the *AWS DataSync User Guide*.

How Amazon DataZone uses AWS Secrets Manager

Amazon DataZone is a data management service that enables you to catalog, discover, govern, share, and analyze your data. You can use data assets from tables and views from an Amazon

Redshift cluster that is crawled using an AWS Glue crawler job. To connect to Amazon Redshift, you provide Amazon DataZone credentials in a Secrets Manager secret. For more information, see [Create a data source for an Amazon Redshift database using a new AWS Glue connection] in the *Amazon DataZone User Guide*.

How AWS Direct Connect uses AWS Secrets Manager

Direct Connect links your internal network to an Direct Connect location over a standard Ethernet fiber-optic cable. With this connection, you can create virtual interfaces directly to public AWS services.

Direct Connect stores a connectivity association key name and connectivity association key pair (CKN/CAK pair) in a [managed secret] with the prefix `directconnect`. The cost of the secret is included with the charge for Direct Connect. To update the secret, you must use Direct Connect rather than Secrets Manager. For more information, see [Associate a MACsec CKN/CAK with a LAG] in the *Direct Connect User Guide*.

How AWS Directory Service uses AWS Secrets Manager

Directory Service provides multiple ways to use Microsoft Active Directory (AD) with other AWS services. You can join an Amazon EC2 instance to your directory using secrets for credentials. For more information, in the *Direct Connect User Guide*, see:

- [Seamlessly join a Linux EC2 instance to your AWS Managed Microsoft AD directory]
- [Seamlessly join a Linux EC2 instance to your AD Connector directory]
- [Seamlessly join a Linux EC2 instance to your Simple AD directory]

How Amazon DocumentDB (with MongoDB compatibility) uses AWS Secrets Manager

Amazon DocumentDB (with MongoDB compatibility) is a fully managed document database service that supports MongoDB workloads. Amazon DocumentDB integrates with Secrets Manager to manage primary user passwords for your clusters, enhancing security and simplifying credential management.

Amazon DocumentDB generates the password, stores it in Secrets Manager, and manages the secret settings. By default, Amazon DocumentDB rotates the secret every seven days, but you can modify the rotation schedule if needed. When you create or modify an Amazon DocumentDB cluster, you can specify that it should manage the primary user password in Secrets Manager. For more information, see Password management with Amazon DocumentDB and Secrets Manager in the *Amazon DocumentDB Developer Guide*.

How AWS Elastic Beanstalk uses AWS Secrets Manager

With AWS Elastic Beanstalk, you can quickly deploy and manage applications in the AWS Cloud without having to learn about the infrastructure that runs those applications. Elastic Beanstalk can launch Docker environments by building an image described in a Dockerfile or pulling a remote Docker image. To authenticate with the online registry that hosts the private repository, Elastic Beanstalk uses a Secrets Manager secret. For more information, see Docker configuration in the *AWS Elastic Beanstalk Developer Guide*.

How Amazon Elastic Container Registry uses AWS Secrets Manager

Amazon Elastic Container Registry (Amazon ECR) is an AWS managed container image registry service that is secure, scalable, and reliable. You can use the Docker CLI, or your preferred client, to push and pull images to and from your repositories. For each upstream registry containing images you want to cache in your Amazon ECR private registry, you must create a pull through cache rule. For upstream registries that require authentication, you must store the credentials in an Secrets Manager secret. You can create the Secrets Manager secret in either the Amazon ECR or Secrets Manager consoles. For more information, see Creating a pull through cache rule in the *Amazon ECR User Guide*.

Amazon Elastic Container Service

Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service that helps you easily deploy, manage, and scale containerized applications. You can inject sensitive data into your containers by referencing Secrets Manager secrets. For more information, see the following pages in the *Amazon Elastic Container Service Developer Guide*:

- Tutorial: Specifying sensitive data using Secrets Manager secrets
- Retrieve secrets programmatically through your application
- Retrieve secrets through environment variables
- Retrieve secrets for logging configuration

Amazon ECS supports FSx for Windows File Server volumes for containers. Amazon ECS uses the credentials stored in a Secrets Manager secret to domain join the Active Directory and attach the

FSx for Windows File Server file system. For more information, see Tutorial: Using FSx for Windows File Server file systems with Amazon ECS and FSx for Windows File Server volumes in the *Amazon Elastic Container Service Developer Guide*.

You can reference container images in private registries outside of AWS that require authentication by using a Secrets Manager secret with the registry credentials. For more information, see Private registry authentication for tasks in the *Amazon Elastic Container Service Developer Guide*.

When you use Amazon ECS Service Connect, Amazon ECS uses Secrets Manager managed secrets to store AWS Private Certificate Authority TLS certificates. The cost of storing the secret is included with the charges for Amazon ECS. To update the secret, you must use Amazon ECS rather than Secrets Manager. For more information, see TLS with Service Connect in the *Amazon Elastic Container Service Developer Guide*.

How Amazon ElastiCache uses AWS Secrets Manager

In ElastiCache you can use a feature called Role-Based Access Control (RBAC) to secure the cluster. You can store these credentials in Secrets Manager. Secrets Manager provides a rotation template for this type of secret. For more information, see Automatically rotating passwords for users in the *Amazon ElastiCache User Guide*.

How AWS Elemental Live uses AWS Secrets Manager

AWS Elemental Live is a real-time video service that lets you create live outputs for broadcast and streaming delivery.

AWS Elemental Live uses a secret ARN to get a secret that contains an encryption key from Secrets Manager. Elemental Live uses the encryption key to encrypt/decrypt the video. For more information, see How delivery from AWS Elemental Live to MediaConnect works at runtime in the *Elemental Live User Guide*.

How AWS Elemental MediaConnect uses AWS Secrets Manager

AWS Elemental MediaConnect is a service that makes it easy for broadcasters and other premium video providers to reliably ingest live video into the AWS Cloud and distribute it to multiple destinations inside or outside the AWS Cloud.

You can use static key encryption to protect your sources, outputs, and entitlements, and you store your encryption key in AWS Secrets Manager. For more information, see Static key encryption in AWS Elemental MediaConnect in the *AWS Elemental MediaConnect User Guide*.

How AWS Elemental MediaConvert uses AWS Secrets Manager

AWS Elemental MediaConvert is a file-based video processing service that provides scalable video processing for content owners and distributors with media libraries of any size. To use MediaConvert to encode Kantar watermarks, you use Secrets Manager to store your Kantar credentials. For more information, see Using Kantar for audio watermarking in AWS Elemental MediaConvert outputs in the *AWS Elemental MediaConvert User Guide*.

How AWS Elemental MediaLive uses AWS Secrets Manager

AWS Elemental MediaLive is a real-time video service that lets you create live outputs for broadcast and streaming delivery. If your organization uses AWS Elemental Link devices with AWS Elemental MediaLive or AWS Elemental MediaConnect, you must deploy the device and configure the device. For more information, see Setting up MediaLive as a trusted entity in the *MediaLive User Guide*.

How AWS Elemental MediaPackage uses AWS Secrets Manager

AWS Elemental MediaPackage is a just-in-time video packaging and origination service that runs in the AWS Cloud. With MediaPackage, you can deliver highly secure, scalable, and reliable video streams to a wide variety of playback devices and content delivery networks (CDNs). For more information, see Secrets Manager access for CDN authorization in the *AWS Elemental MediaPackage User Guide*.

How AWS Elemental MediaTailor uses AWS Secrets Manager

AWS Elemental MediaTailor is a scalable ad insertion and channel assembly service that runs in the AWS Cloud.

MediaTailor supports Secrets Manager access token authentication to your source locations. With Secrets Manager access token authentication, MediaTailor uses a Secrets Manager secret to authenticate requests to your origin. For more information, see Configuring AWS Secrets Manager access token authentication in the *AWS Elemental MediaTailor User Guide*.

How Amazon EMR uses Secrets Manager

Amazon EMR is a platform that simplifies running big data frameworks, such as Apache Hadoop and Apache Spark, on AWS to process and analyze vast amounts of data. When you use these frameworks and related open-source projects such as Apache Hive and Apache Pig, you can process data for analytics and business intelligence workloads. You can also use Amazon EMR to transform and move large amounts of data into and out of other AWS data stores and databases, such as Amazon S3 and Amazon DynamoDB.

How Amazon EMR running on Amazon EC2 uses Secrets Manager

When you create a cluster in Amazon EMR, you can provide application configuration data to the cluster with a secret in Secrets Manager. For more information, see Store sensitive configuration data in Secrets Manager in the *Amazon EMR Management Guide*.

In addition, when you create an EMR Notebook, you can store your private Git-based registry credentials using Secrets Manager. For more information, see Add a Git-based Repository to Amazon EMR in the *Amazon EMR Management Guide*.

How EMR Serverless uses Secrets Manager

EMR Serverless provides a serverless runtime environment to simplify the operation of analytics applications so that you don't have to configure, optimize, secure, or operate clusters.

You can store your data in AWS Secrets Manager and then use the secret ID in your EMR Serverless configurations. This way, you don't pass sensitive configuration data in plain text and expose it to external APIs.

For more information, see Secrets Manager for data protection with EMR Serverless in the *Amazon EMR Serverless User Guide*.

How Amazon EventBridge uses AWS Secrets Manager

Amazon EventBridge is a serverless event bus service that you can use to connect your applications with data from a variety of sources.

When you create an Amazon EventBridge API destination, EventBridge stores the connection for it in a Secrets Manager managed secret with the prefix `events`. The cost of storing the secret is included with the charge for using an API destination. To update the secret, you must use EventBridge rather than Secrets Manager. For more information, see API destinations in the *Amazon EventBridge User Guide*.

How Amazon FSx uses AWS Secrets Manager secrets

Amazon FSx for Windows File Server provides fully managed Microsoft Windows file servers, backed by a fully native Windows file system. When you create or manage file shares, you can pass credentials from an AWS Secrets Manager secret. For more information, see File shares and Migrating file share configurations to Amazon FSx in the *Amazon FSx for Windows File Server User Guide*.

How AWS Glue DataBrew uses AWS Secrets Manager

AWS Glue DataBrew is a visual data preparation tool that you can use to clean and normalize data without writing any code. In DataBrew, a set of data transformation steps is called a recipe. AWS Glue DataBrew provides the `DETERMINISTIC_DECRYPT`, `DETERMINISTIC_ENCRYPT`, and `CRYPTOGRAPHIC_HASH` recipe steps to perform transformations on personally identifiable information (PII) in a dataset, which use an encryption key stored in a Secrets Manager secret. If you use the DataBrew *default secret* to store the encryption key, DataBrew creates a managed secret with the prefix `databrew`. The cost of storing the secret is included with the charge for using DataBrew. If you create a new secret to store the encryption key, DataBrew creates a secret with the prefix `AwsGlueDataBrew`. You are charged for that secret.

How AWS Glue Studio uses AWS Secrets Manager

AWS Glue Studio is a graphical interface that makes it easy to create, run, and monitor extract, transform, and load (ETL) jobs in AWS Glue. You can use Amazon OpenSearch Service as a data store for your extract, transform, and load (ETL) jobs by configuring the Elasticsearch Spark Connector in AWS Glue Studio. To connect to the OpenSearch cluster, you can use a secret in Secrets Manager. For more information, see Tutorial: Using the AWS Glue Connector for Elasticsearch in the *AWS Glue Developer Guide*.

How AWS IoT SiteWise uses AWS Secrets Manager

AWS IoT SiteWise is a managed service that lets you collect, model, analyze, and visualize data from industrial equipment at scale. You can use the AWS IoT SiteWise console to create a gateway. Then add data sources, local servers or industrial equipment that are connected to gateways. If your source requires authentication, use a secret to authenticate. For more information, see Configuring data source authentication in the *AWS IoT SiteWise User Guide*.

How Amazon Kendra uses AWS Secrets Manager

Amazon Kendra is a highly accurate and intelligent search service that enables your users to search unstructured and structured data using natural language processing and advanced search algorithms.

You can index documents stored in a database by specifying a secret that contains credentials for the database. For more information, see Using a database data source in the *Amazon Kendra User Guide*.

How Amazon Kinesis Video Streams uses AWS Secrets Manager

You can use Amazon Kinesis Video Streams to connect to IP cameras on customer premises, locally record and store video from the cameras, and stream videos to the cloud for long-term storage, playback, and analytical processing. To record and upload media from IP cameras, you deploy the Kinesis Video Streams Edge Agent to AWS IoT Greengrass. You store the credentials required to access the media files that are streamed to the camera in an Secrets Manager secret. For more information, see Deploy the Amazon Kinesis Video Streams Edge Agent to AWS IoT Greengrass in the *Amazon Kinesis Video Streams Developer Guide*.

How AWS Launch Wizard uses AWS Secrets Manager

AWS Launch Wizard for Active Directory is a service that applies AWS Cloud application best practices to guide you through setting up a new Active Directory infrastructure, or adding domain controllers to an existing infrastructure, either in the AWS Cloud or on premises.

AWS Launch Wizard requires domain administrator credentials to be added to Secrets Manager to join your domain controllers to Active Directory. For more information, see Set up for AWS Launch Wizard for Active Directory in the *AWS Launch Wizard User Guide*.

How Amazon Lookout for Metrics uses AWS Secrets Manager

Amazon Lookout for Metrics is a service that finds anomalies in your data, determines their root causes, and enables you to quickly take action. You can use Amazon Redshift or Amazon RDS as a datasource for an Lookout for Metrics detector. To configure the datasource, you use a secret that contains the database password. For more information, see Using Amazon RDS with Lookout for Metrics and Using Amazon Redshift with Lookout for Metrics in the *Amazon Lookout for Metrics Developer Guide*.

How Amazon Managed Grafana uses AWS Secrets Manager

Amazon Managed Grafana is a fully managed and secure data visualization service that you can use to instantly query, correlate, and visualize operational metrics, logs, and traces from multiple sources. When you use Amazon Redshift as a data source, you can provide Amazon Redshift credentials by using an AWS Secrets Manager secret. For more information, see Configuring Amazon Redshift in the *Amazon Managed Grafana User Guide*.

How AWS Managed Services uses AWS Secrets Manager

AWS Managed Services is an enterprise service that provides ongoing management of your AWS infrastructure. AMS Self-Service Provisioning (SSP) mode provides full access to native AWS service and API Capabilities in AMS managed accounts. For information about how to request access to Secrets Manager in AMS, see AWS Secrets Manager (AMS self-service provisioning) in the *AMS Advanced User Guide*.

How Amazon Managed Streaming for Apache Kafka uses AWS Secrets Manager

Amazon Managed Streaming for Apache Kafka (Amazon MSK) is a fully managed service that enables you to build and run applications that use Apache Kafka to process streaming data. You can control access to your Amazon MSK clusters using usernames and passwords that are stored and secured using AWS Secrets Manager. For more information, see Username and password authentication with AWS Secrets Manager in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

How Amazon Managed Workflows for Apache Airflow uses AWS Secrets Manager

Amazon Managed Workflows for Apache Airflow is a managed orchestration service for Apache Airflow that makes it easier to setup and operate end-to-end data pipelines in the cloud at scale.

You can configure an Apache Airflow connection using a Secrets Manager secret. For more information, see Configuring an Apache Airflow connection using a Secrets Manager secret and Using a secret key in AWS Secrets Manager for an Apache Airflow variable in the *Amazon Managed Workflows for Apache Airflow User Guide*.

AWS Marketplace

When you use AWS Marketplace Quick Launch, AWS Marketplace distributes your software along with the license key. AWS Marketplace stores the license key in your account as a Secrets Manager managed secret. The cost of storing the secret is included with the charges for AWS Marketplace.

To update the secret, you must use AWS Marketplace rather than Secrets Manager. For more information, see Configure Quick Launch in the *AWS Marketplace Seller Guide*.

How AWS Migration Hub uses AWS Secrets Manager

AWS Migration Hub provides a single location to track migration tasks across multiple AWS tools and partner solutions.

AWS Migration Hub Orchestrator simplifies and automates the migration of servers and enterprise applications to AWS. Migration Hub Orchestrator uses a secret for the connection information to your source server. For more information, in the *AWS Migration Hub Orchestrator User Guide*, see:

- Migrate SAP NetWeaver applications to AWS
- Rehost applications on Amazon EC2

Migration Hub Strategy Recommendations offers migration and modernization strategy recommendations for viable transformation paths for your applications. Strategy Recommendations can analyze SQL Server databases, using a secret for the connection information. For more information, see Strategy Recommendations database analysis.

How AWS Panorama uses Secrets Manager

AWS Panorama is a service that brings computer vision to your on-premises camera network. You use AWS Panorama to register an appliance, update its software, and deploy applications to it. When you register a video stream as a data source for your application, if the stream is password protected, AWS Panorama stores the credentials for it in a Secrets Manager secret. For more information, see Managing camera streams in AWS Panorama in the *AWS Panorama Developer Guide*.

How AWS Parallel Computing Service uses AWS Secrets Manager

AWS Parallel Computing Service (AWS PCS) is a managed service that makes it easier to run and scale high performance computing (HPC) and distributed machine learning workloads on AWS.

To connect to the cluster job scheduler, AWS PCS creates a managed secret with the prefix `pcs` to store the scheduler key. The cost of storing the secret is included with the charge for AWS PCS. AWS PCS automatically deletes the secret when you delete your AWS PCS cluster. For more information, see Working with cluster secrets in AWS PCS in the *AWS PCS User Guide*.

Important

Don't modify or delete AWS PCS cluster secrets.

How AWS ParallelCluster uses AWS Secrets Manager

AWS ParallelCluster is an open source cluster management tool that you can use to deploy and manage high performance computing (HPC) clusters in the AWS Cloud. You can create a multiple user environment that includes an AWS ParallelCluster that's integrated with an AWS Managed Microsoft AD (Active Directory). The AWS ParallelCluster uses a Secrets Manager secret for validating logins to Active Directory. For more information, see Integrating Active Directory in the *AWS ParallelCluster User Guide*.

How Amazon Q uses Secrets Manager

To authenticate Amazon Q to access your data source, you provide your data source access credentials to Amazon Q using an Secrets Manager secret. If you use the console, you can choose to create a new secret or use an existing one. For more information, see Concepts – Authentication in the *Amazon Q Developer Guide*.

How Amazon OpenSearch Ingestion uses Secrets Manager

Amazon OpenSearch Ingestion is a fully managed, serverless data collector that streams real-time logs, metrics, and trace data to Amazon OpenSearch Service domains and OpenSearch Serverless collections. You can use OpenSearch Ingestion pipelines with Secrets Manager to securely manage your credentials. For more information, see:

- Using an OpenSearch Ingestion pipeline with Atlassian Services
- Using an OpenSearch Ingestion pipeline with Amazon DocumentDB
- Using an OpenSearch Ingestion pipeline with Confluent Cloud Kafka
- Using an OpenSearch Ingestion pipeline with Kafka
- Migrating data from self-managed OpenSearch clusters using Amazon OpenSearch Ingestion

How AWS OpsWorks for Chef Automate uses AWS Secrets Manager

OpsWorks is a configuration management service that helps you configure and operate applications in a cloud enterprise by using OpsWorks for Puppet Enterprise or AWS OpsWorks for Chef Automate.

When you create a new server in AWS OpsWorks CM, OpsWorks CM stores information for the server in a Secrets Manager managed secret with the prefix `opsworks-cm`. The cost of the secret is included in the charge for OpsWorks. For more information, see Integration with AWS Secrets Manager in the *OpsWorks User Guide*.

How Amazon Quick Suite uses AWS Secrets Manager

Amazon Quick Suite is a cloud-scale business intelligence (BI) service you can use for analytics, data visualization, and reporting. You can use a variety of data sources in Quick Suite. If you store database credentials in Secrets Manager secrets, Quick Suite can use those secrets to connect to the databases. For more information, see Using AWS Secrets Manager secrets in place of database credentials in Amazon Quick Suite in the *Amazon Quick Suite User Guide*.

How Amazon RDS uses AWS Secrets Manager

Amazon Relational Database Service (Amazon RDS) is a web service that makes it easier to set up, operate, and scale a relational database in the AWS Cloud.

To manage master user credentials for Amazon Relational Database Service (Amazon RDS), including Aurora, Amazon RDS can create a managed secret for you. You are charged for that secret. Amazon RDS also manages rotation for these credentials. For more information, see Password management with Amazon RDS and AWS Secrets Manager in the *Amazon RDS User Guide*.

For other Amazon RDS credentials, see *Create secrets*.

When you use the Amazon RDS query editor to connect to a database, you can store credentials for the database in Secrets Manager. For more information, see Using the query editor in the *Amazon RDS User Guide*.

How Amazon Redshift uses AWS Secrets Manager

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud.

To manage admin credentials for Amazon Redshift, Amazon Redshift can create a managed secret for you. You are charged for that secret. Amazon Redshift also manages rotation for these credentials. For more information, see Managing Amazon Redshift admin passwords using AWS Secrets Manager in the *Amazon Redshift Management Guide*.

For other Amazon Redshift credentials, see *Create secrets*.

When you call the Amazon Redshift Data API, you can pass credentials for the cluster by using a secret in Secrets Manager. For more information, see Using the Amazon Redshift Data API.

When you use the Amazon Redshift query editor to connect to a database, Amazon Redshift can store your credentials in a Secrets Manager secret with the prefix `redshiftqueryeditor`. You are charged for that secret. For more information, see Querying a database using the query editor in the *Amazon Redshift Management Guide*.

For query editor v2, see the section called "Amazon Redshift query editor v2".

Amazon Redshift query editor v2

Amazon Redshift query editor v2 is a web-based SQL client application that you can use to author and run queries on your Amazon Redshift data warehouse. When you use the Amazon Redshift query editor v2 to connect to a database, Amazon Redshift can store your credentials in a Secrets Manager managed secret with the prefix `sqlworkbench`. The cost of storing the secret is included with the charge for using Amazon Redshift. To update the secret, you must use Amazon Redshift rather than Secrets Manager. For more information, see Working with query editor v2 in the *Amazon Redshift Management Guide*.

For the previous query editor, see the section called "Amazon Redshift".

How Amazon SageMaker AI uses AWS Secrets Manager

SageMaker AI is a fully managed machine learning service. With SageMaker AI, data scientists and developers can quickly and easily build and train machine learning models, and then directly deploy them into a production-ready hosted environment. It provides an integrated Jupyter authoring notebook instance for easy access to your data sources for exploration and analysis, so you don't have to manage servers.

You can associate Git repositories with your Jupyter notebook instances to save your notebooks in a source control environment that persists even if you stop or delete your notebook instance. You can manage your private repositories credentials using Secrets Manager. For more information, see Associate Git Repositories with Amazon SageMaker Notebook Instances in the *Amazon SageMaker AI Developer Guide*.

To import data from Databricks, Data Wrangler stores your JDBC URL in Secrets Manager. For more information, see Import data from Databricks (JDBC).

To import data from Snowflake, Data Wrangler stores your credentials in a Secrets Manager secret. For more information, see Import data from Snowflake.

How AWS Schema Conversion Tool uses AWS Secrets Manager

You can use the AWS Schema Conversion Tool (AWS SCT) to convert your existing database schema from one database engine to another. You can convert relational OLTP schema, or data warehouse schema. Your converted schema is suitable for an Amazon Relational Database Service (Amazon RDS) MySQL, MariaDB, Oracle, SQL Server, PostgreSQL DB, an Amazon Aurora DB cluster, or an Amazon Redshift cluster. The converted schema can also be used with a database on an Amazon Elastic Compute Cloud instance or stored as data on an S3 bucket.

When you convert a database schema, AWS SCT can use database credentials that you store in AWS Secrets Manager. For more information, see Using AWS Secrets Manager in the AWS SCT user interface in the *AWS Schema Conversion Tool User Guide*.

How Amazon Timestream for InfluxDB uses AWS Secrets Manager

Timestream for InfluxDB is a managed time-series database engine that makes it easy for you to run InfluxDB databases on AWS for real-time time-series applications using open-source APIs. With Timestream for InfluxDB, you can set up, operate, and scale time-series workloads that can answer queries with single-digit millisecond query response time.

When you create a Timestream for InfluxDB database, Timestream automatically creates a secret to store the admin credentials. For more information, see How Timestream for InfluxDB uses secrets in the *Timestream Developer Guide*.

How AWS Toolkit for JetBrains uses AWS Secrets Manager

The AWS Toolkit for JetBrains is an open source plugin for the integrated development environments (IDEs) from JetBrains. The toolkit makes it easier for developers to develop, debug, and deploy serverless applications that use AWS. When connecting to an Amazon Redshift cluster using the toolkit, you can authenticate using a Secrets Manager secret. For more information, see Accessing Amazon Redshift clusters in the *AWS Toolkit for JetBrains User Guide*.

How AWS Transfer Family uses AWS Secrets Manager secrets

AWS Transfer Family is a secure transfer service that enables you to transfer files into and out of AWS storage services.

Transfer Family now supports using Basic authentication for servers that use the Applicability Statement 2 (AS2) protocol. You can create a new Secrets Manager secret or choose an existing secret for your credentials. For more information, see Basic authentication for AS2 connectors in the *AWS Transfer Family User Guide*.

To authenticate Transfer Family users, you can use AWS Secrets Manager as an identity provider. For more information, see Working with custom identity providers in the *AWS Transfer Family User Guide* and the blog article Enable password authentication for AWS Transfer Family using AWS Secrets Manager.

You can use Pretty Good Privacy (PGP) decryption with the files that Transfer Family processes with workflows. To use decryption in a workflow step, you provide a PGP key that you manage in Secrets Manager. For more information, see Generate and manage PGP keys in the *AWS Transfer Family User Guide*.

How AWS Wickruses AWS Secrets Manager secrets

AWS Wickr is an end-to-end encrypted service that helps organizations and government agencies to communicate securely through one-to-one and group messaging, voice and video calling, file sharing, screen sharing, and more. You can automate workflows using Wickr data retention bots. If the bot will have access to AWS services, then you should create a Secrets Manager secret to store the bot credentials. For more information, see Start the data retention bot in the *AWS Wickr Administration Guide*.

Using AWS Secrets Manager managed external secrets to manage Third Party secrets

Managed external secrets is a new secret type in AWS Secrets Manager that enables you to store and automatically rotate credentials from integration partners. This feature eliminates the need to create and maintain custom AWS Lambda functions for rotating integration partner secrets. For a complete list of all onboarded partners see Integration Partners.

When you build applications on AWS, your workloads often need to interact with third-party applications through secure credentials such as API keys, OAuth tokens, or credential pairs. Previously, you had to develop custom approaches to secure and manage these credentials, including building complex rotation Lambda functions that were unique to each application and required ongoing maintenance.

Managed external secrets provides a standardized approach for storing third-party credentials in a predefined format prescribed by each partner. The feature includes automatic rotation that is enabled (by default on the console) during secret creation, complete transparency and user controls for secret management workflows, and the full feature set offered by Secrets Manager including fine-grained permissions management, observability, governance, compliance, disaster recovery, and monitoring controls.

Key features

Managed external secrets offers several key capabilities that simplify third-party credential management:

- **Lambda-free managed rotation** eliminates the overhead of creating and managing custom rotation functions. When you create an external, rotation is automatically enabled with no Lambda functions deployed in your account.
- **Predefined secret formats** ensure that secrets can be properly associated with the integration partner and include the metadata needed for rotation. Each partner defines the required format.
- **Integrated partner ecosystem** provides support for multiple partners through a standardized onboarding process. Partners integrate directly with Secrets Manager to offer programmatic guidance for secret creation and managed rotation capabilities.
- **Complete auditability** maintains full transparency through AWS CloudTrail logging for all rotation activities, secret value updates, and management operations.

Managed external secrets Partners

Secrets Manager natively integrates with third party applications to rotate secrets held by the partner. Each partner defines the metadata and secret value fields required to rotate the secrets.

The secret value contains fields that are required for connecting with your third party client and are stored during the CreateSecret call. The rotation metadata holds the fields that are used to update the secret during rotation and are used in the RotateSecret call. These fields will be defined by the integration partner to allow managed rotation flows.

For rotation to function properly, you must provide Secrets Manager with specific permissions to manage the secret lifecycle. For more information see Security and Permissions

The following topics include a description of each of the metadata fields required to rotate the secret as well as a description of each of the fields required in the Secrets Manager secret to rotate.

Topics

Integration Partner	Secret type
Salesforce	SalesforceClientSecret
BigID	BigIDClientSecret
Snowflake	SnowflakeKeyPairAuthentication

Salesforce Client Secret

Secret Value Fields

The following are the fields that must be contained in the Secrets Manager secret:

```
{
  "consumerKey": "client ID",
  "consumerSecret": "client secret",
  "baseUri": "https://domain.my.salesforce.com",
  "appId": "app ID",
  "consumerId": "consumer ID"
}
```

consumerKey

The consumer key, also known as the client ID, is the credential identifier for the OAuth 2.0 credentials. You can retrieve the consumer key directly from the Salesforce External Client App Manager OAuth settings.

consumerSecret

The consumer secret, also known as the client secret, is the private password used with the consumer key to authenticate using the OAuth 2.0 client credentials flow. You can retrieve the consumer secret directly from the Salesforce External Client App Manager OAuth settings..

baseUri

The base URI is your Salesforce Org's base URL used to interact with Salesforce APIs. This takes the form of the following example: `https://domainName.my.salesforce.com`.

appId

The App ID is the identifier for your Salesforce External Client Application (ECA). You can retrieve this by calling the Salesforce OAuth Usage endpoint. It must begin with `0x` and contain only alphanumeric characters. This field refers to the external_client_app_identifier in the Salesforce rotation guide.

consumerId

The consumer ID is the identifier for your Salesforce External Client Application (ECA) consumer. You can retrieve this by calling the Salesforce OAuth Credentials by App ID endpoint. This field refers to the consumer_id in the Salesforce rotation guide.

Secret Metadata Fields

The following are the metadata fields required to rotate a secret held by Salesforce.

```
{
  "apiVersion": "v65.0",
  "adminSecretArn": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SalesforceClientSecret"
}
```

apiVersion

The Salesforce API version is your Salesforce organization's API version. The version should be at least v65.0. It must be in the format *vXX.X* where *X* is a numeric character.

adminSecretArn

(Optional) The admin secret ARN is the Amazon Resource Name (ARN) for the secret that contains the administrative OAuth credentials that are to used to rotate this Salesforce client secret. At a minimum the admin secret should contain a consumerKey and consumerSecret value within the secret structure. It is an optional field and if omitted, during rotation Secrets Manager will use the OAuth credentials within this secret to authenticate with Salesforce.

Usage Flow

Customers storing Salesforce Secrets in AWS Secrets Manager have an option to rotate a secret with the credentials stored in the same secret or use the credentials in the Admin secret for rotation. You can create your secret using the CreateSecret call with the secret value containing the fields mentioned above and secret type as SalesforceClientSecret. The rotation configurations can be set using a RotateSecret call. This call requires the specification of the metadata fields as in the example above - If you opt for a rotation using credentials in the same secret, you can skip the adminSecretArn field. Additionally, customers must provide a role ARN in the RotateSecret call which grants the service the required permissions to rotate the secret. For an example of a permissions policy see Security and Permissions.

For customers opting to rotate their secrets using a seperate set of credentials (stored in an Admin Secret), be sure to create the Admin Secret in AWS Secrets Manager following the exact same steps as your consumer secret. You must provide the ARN of this Admin Secret in the rotation metadata in a RotateSecret call for your consumer secret.

The rotation logic follows the guidance provided by Salesforce.

Big ID Refresh Token

Secret Value Fields

The following are the fields that must be contained in the Secrets Manager secret:

```
{
  "hostname": "Host Name",
```

```
  "refreshToken": "Refresh Token"
}
```

hostname

This is the hostname where your BigID instance is hosted. You must enter the fully qualified domain name of your instance.

refreshToken

The JWT user refresh token generated in the BigID Console via Administration → Access Management → Select User → Generate Token → Save

Usage Flow

You can create your secret using the CreateSecret call with the secret value containing the fields mentioned above and secret type as BigIDClientSecret. The rotation configurations can be set using a RotateSecret call. You must also provide a role ARN in the RotateSecret call which grants the service the required permissions to rotate the secret. For example of a permissions policy see Security and Permissions. Note that the rotation metadata field can be left empty for this partner.

Snowflake Key Pair

Secret Value Fields

The following are the fields that must be contained in the Secrets Manager secret:

```
{
  "account": "Your Account Identifier",
  "user": "Your user name",
  "privateKey": "Your private Key",
  "publicKey": "Your public Key",
  "passphrase": "Your Passphrase"
}
```

user

The Snowflake username associated with this key-pair authentication. This user must be configured in Snowflake to accept key-pair authentication, and the public key must be assigned to this user's profile.

account

Your Snowflake account identifier used to establish the connection. This can be extracted from your Snowflake URL (the portion before .snowflakecomputing.com)

privateKey

The RSA private key in PEM format used for authentication. The BEGIN/END markers are optional.

publicKey

The public key counterpart in PEM format corresponding to the private key. The BEGIN/END markers are optional.

passphrase

(Optional) This field refers to the passphrase used to decrypt the encrypted private key.

Secret Metadata Fields

The following are the metadata fields for Snowflake:

```
{
  "cryptographicAlgorithm": "Your Cryptographic algorithm",
  "encryptPrivateKey": "True/False"
}
```

cryptographicAlgorithm

(Optional) This refers to the algorithm used for key generation. You have a choice of 3 algorithms: RS256 | RS384 | RS512 . This field is optional and the default algorithm chosen is RS256.

encryptPrivateKey

(Optional) This field can be used to choose if you want to encrypt your private key. It is false by default. The passphrase for encryption is randomly generated.

Usage Flow

You can create your secret using the [CreateSecret](#) call with the secret value containing the fields mentioned above and secret type as SnowflakeKeyPairAuthentication. The rotation configurations can be set using a RotateSecret call. You can optionally provide the secret metadata field(s) based on your requirement. You must also provide a role ARN in the RotateSecret call which grants the service the required permissions to rotate the secret. For example of a permissions policy see Security and Permissions. Note that the rotation metadata field can be left empty for this partner.

Security and permissions

Managed external secrets does not require you to share admin-level privileges of your third party application accounts with AWS. Instead, the rotation process uses credentials and metadata you provide to make authorized API calls to the third party application for credential updates and validation.

Managed external secrets maintain the same security standards as other Secrets Manager secret types. Secret values are encrypted at rest using your KMS keys and in transit using TLS. Access to secrets is controlled through IAM policies and resource-based policies. When using a Customer Managed Key to encrypt your secret, you will need to update the IAM policy of the rotation role and CMK trust policy to provide the required permissions to ensure successful rotation.

For rotation to function properly, you must provide Secrets Manager with specific permissions to manage the secret lifecycle. These permissions can be scoped to individual secrets and follow the principle of least privilege. The rotation role you provide is validated during setup and used exclusively for rotation operations.

You can restrict the IP ingress to your external resource by only allowing the AWS IP ranges for EC2 in the region where your secret exists. This list of IP ranges can change so you should refresh your ingress rules periodically.

AWS Secrets Manager also offers single touch solutions to create the IAM policy with the permissions necessary to manage the secret when creating the secret through the Secrets Manager console. The permissions for this role are scoped down for each integration partner in each region.

Example Permissions Policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowRotationAccess",
      "Action": [
        "secretsmanager:DescribeSecret",
```

```
        "secretsmanager:GetSecretValue",
        "secretsmanager:PutSecretValue",
        "secretsmanager:UpdateSecretVersionStage"
      ],
      "Resource": "*",
      "Effect": "Allow",
      "Condition": {
        "StringEquals": {
          "secretsmanager:resource/Type": "SalesforceClientSecret"
        }
      }
    },
    {
      "Sid": "AllowPasswordGenerationAccess",
      "Action": [
        "secretsmanager:GetRandomPassword"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```

Note: The list of secret types that are available for secretsmanager:resource/Type can be found in Integration Partners.

Example Trust Policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "SecretsManagerPrincipalAccess",
      "Effect": "Allow",
      "Principal": {
        "Service": "secretsmanager.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
```

```
          "aws:SourceArn": "arn:aws:secretsmanager:us-east-1:111122223333:secret:*"
        }
      }
    }
  ]
}
```

Monitor and troubleshoot managed external secrets

Managed external secrets provide comprehensive monitoring capabilities through AWS CloudTrail logs and Amazon CloudWatch metrics. All rotation activities are logged with detailed information about success, failure, and any errors encountered during the process.

Common issues in the rotation workflow include an incorrect configuration of role permissions or the secret value. Failure to set these fields is the format specified by the integration partners can cause rotation failures, as the service will be unable to access the secret or connect with the integration partner client to update the secret. Other issues could be network connectivity problems, credential expiration, or partner service availability. The managed rotation service includes retry logic and error handling to maximize reliability

You can monitor rotation schedules, success rates, and performance metrics through Amazon CloudWatch. You can configure custom alarms through event bridge to alert you of rotation failures or other issues that require attention.

Migrating existing secrets

You have an option to migrate your existing partner secrets to managed external secrets. This can be done with an UpdateSecret call. You must update the secret value and metadata as mentioned in the guide. If you already have custom rotation logic set up for these secrets, you must first cancel the rotation using a CancelRotateSecret call.

Limitations and considerations

Managed external secrets does not support ephemeral secrets with lifespans less than four hours. Secrets associated with public key infrastructure certificates are also not supported.

The managed external secrets are supported only for partners that have onboarded with AWS Secrets Manager. For a complete list, see Integration Partners. Don't see your partner on the list? Tell them to Onboard to AWS Secrets Manager

If you update or rotate secret values directly from the partner client service outside of the Secrets Manager rotation engine, the synchronization between systems may break. While Secrets Manager provides console warnings and programmatic prevention for manual secret value updates, you can still modify values directly in your third party application. To re-establish synchronization after out-of-band updates, you must update the secret value to reflect the correct secret and then invoke the RotateSecret API to ensure continued successful rotations.

Create AWS Secrets Manager secrets in AWS CloudFormation

You can create secrets in a CloudFormation stack by using the `AWS::SecretsManager::Secret` resource in a CloudFormation template, as shown in Create a secret.

To create an admin secret for Amazon RDS or Aurora, we recommend you use `ManageMasterUserPassword` in `AWS::RDS::DBCluster`. Then Amazon RDS creates the secret and manages rotation for you. For more information, see Managed rotation.

For Amazon Redshift and Amazon DocumentDB credentials, first create a secret with a password generated by Secrets Manager, and then use a dynamic reference to retrieve the username and password from the secret to use as credentials for a new database. Next, use the `AWS::SecretsManager::SecretTargetAttachment` resource to add details about the database to the secret that Secrets Manager needs to rotate the secret. Finally, to turn on automatic rotation, use the `AWS::SecretsManager::RotationSchedule` resource and provide a rotation function and a schedule. See the following examples:

- Create a secret with Amazon Redshift credentials
- Create a secret with Amazon DocumentDB credentials

To attach a resource policy to your secret, use the `AWS::SecretsManager::ResourcePolicy` resource.

For information about creating resources with CloudFormation, see Learn template basics in the CloudFormation User Guide. You can also use the AWS Cloud Development Kit (AWS CDK). For more information, see AWS Secrets Manager Construct Library.

Create an AWS Secrets Manager secret with CloudFormation

This example creates a secret named **`CloudFormationCreatedSecret-`*`a1b2c3d4e5f6`***. The secret value is the following JSON, with a 32-character password that is generated when the secret is created.

```
{
```

```
    "password": "EXAMPLE-PASSWORD",
    "username": "saanvi"
}
```

This example uses the following CloudFormation resource:

- AWS::SecretsManager::Secret

For information about creating resources with CloudFormation, see Learn template basics in the CloudFormation User Guide.

JSON

```
{
    "Resources": {
        "CloudFormationCreatedSecret": {
            "Type": "AWS::SecretsManager::Secret",
            "Properties": {
                "Description": "Simple secret created by CloudFormation.",
                "GenerateSecretString": {
                    "SecretStringTemplate": "{\"username\": \"saanvi\"}",
                    "GenerateStringKey": "password",
                    "PasswordLength": 32
                }
            }
        }
    }
}
```

YAML

```
Resources:
  CloudFormationCreatedSecret:
    Type: 'AWS::SecretsManager::Secret'
    Properties:
      Description: Simple secret created by CloudFormation.
      GenerateSecretString:
        SecretStringTemplate: '{"username": "saanvi"}'
        GenerateStringKey: password
        PasswordLength: 32
```

Create an AWS Secrets Manager secret with automatic rotation and an Amazon RDS MySQL DB instance with CloudFormation

To create an admin secret for Amazon RDS or Aurora, we recommend you use `ManageMasterUserPassword`, as shown in the example *Create a Secrets Manager secret for a master password* in `AWS::RDS::DBCluster`. Then Amazon RDS creates the secret and manages rotation for you. For more information, see Managed rotation.

Create an AWS Secrets Manager secret and an Amazon Redshift cluster with CloudFormation

To create an admin secret for Amazon Redshift, we recommend you use the examples on `AWS::Redshift::Cluster` and `AWS::RedshiftServerless::Namespace`.

Create an AWS Secrets Manager secret and an Amazon DocumentDB instance with CloudFormation

This example creates a secret and an Amazon DocumentDB instance using the credentials in the secret as the user and password. The secret has a resource-based policy attached that defines who can access the secret. The template also creates a Lambda rotation function from the Rotation function templates and configures the secret to automatically rotate between 8:00 AM and 10:00 AM UTC on the first day of every month. As a security best practice, the instance is in an Amazon VPC.

This example uses the following CloudFormation resources for Secrets Manager:

- `AWS::SecretsManager::Secret`
- `AWS::SecretsManager::SecretTargetAttachment`
- `AWS::SecretsManager::RotationSchedule`

For information about creating resources with CloudFormation, see Learn template basics in the CloudFormation User Guide.

JSON

```
{
```

```
    "AWSTemplateFormatVersion":"2010-09-09",
    "Transform":"AWS::SecretsManager-2020-07-23",
    "Resources":{
       "TestVPC":{
          "Type":"AWS::EC2::VPC",
          "Properties":{
             "CidrBlock":"10.0.0.0/16",
             "EnableDnsHostnames":true,
             "EnableDnsSupport":true
          }
       },
       "TestSubnet01":{
          "Type":"AWS::EC2::Subnet",
          "Properties":{
             "CidrBlock":"10.0.96.0/19",
             "AvailabilityZone":{
                "Fn::Select":[
                   "0",
                   {
                      "Fn::GetAZs":{
                         "Ref":"AWS::Region"
                      }
                   }
                ]
             },
             "VpcId":{
                "Ref":"TestVPC"
             }
          }
       },
       "TestSubnet02":{
          "Type":"AWS::EC2::Subnet",
          "Properties":{
             "CidrBlock":"10.0.128.0/19",
             "AvailabilityZone":{
                "Fn::Select":[
                   "1",
                   {
                      "Fn::GetAZs":{
                         "Ref":"AWS::Region"
                      }
                   }
                ]
             },
```

```
            "VpcId":{
               "Ref":"TestVPC"
            }
         }
      },
      "SecretsManagerVPCEndpoint":{
         "Type":"AWS::EC2::VPCEndpoint",
         "Properties":{
            "SubnetIds":[
               {
                  "Ref":"TestSubnet01"
               },
               {
                  "Ref":"TestSubnet02"
               }
            ],
            "SecurityGroupIds":[
               {
                  "Fn::GetAtt":[
                     "TestVPC",
                     "DefaultSecurityGroup"
                  ]
               }
            ],
            "VpcEndpointType":"Interface",
            "ServiceName":{
               "Fn::Sub":"com.amazonaws.${AWS::Region}.secretsmanager"
            },
            "PrivateDnsEnabled":true,
            "VpcId":{
               "Ref":"TestVPC"
            }
         }
      },
      "MyDocDBClusterRotationSecret":{
         "Type":"AWS::SecretsManager::Secret",
         "Properties":{
            "GenerateSecretString":{
               "SecretStringTemplate":"{\"username\": \"someadmin\",\"ssl\": true}",
               "GenerateStringKey":"password",
               "PasswordLength":16,
               "ExcludeCharacters":"\"@/\\"
            },
            "Tags":[
```

```
                {
                   "Key":"AppName",
                   "Value":"MyApp"
                }
             ]
          }
       },
       "MyDocDBCluster":{
          "Type":"AWS::DocDB::DBCluster",
          "Properties":{
             "DBSubnetGroupName":{
                "Ref":"MyDBSubnetGroup"
             },
             "MasterUsername":{
                "Fn::Sub":"{{resolve:secretsmanager:
${MyDocDBClusterRotationSecret}::username}}"
             },
             "MasterUserPassword":{
                "Fn::Sub":"{{resolve:secretsmanager:
${MyDocDBClusterRotationSecret}::password}}"
             },
             "VpcSecurityGroupIds":[
                {
                   "Fn::GetAtt":[
                      "TestVPC",
                      "DefaultSecurityGroup"
                   ]
                }
             ]
          }
       },
       "DocDBInstance":{
          "Type":"AWS::DocDB::DBInstance",
          "Properties":{
             "DBClusterIdentifier":{
                "Ref":"MyDocDBCluster"
             },
             "DBInstanceClass":"db.r5.large"
          }
       },
       "MyDBSubnetGroup":{
          "Type":"AWS::DocDB::DBSubnetGroup",
          "Properties":{
             "DBSubnetGroupDescription":"",
```

```
            "SubnetIds":[
               {
                  "Ref":"TestSubnet01"
               },
               {
                  "Ref":"TestSubnet02"
               }
            ]
         }
      },
      "SecretDocDBClusterAttachment":{
         "Type":"AWS::SecretsManager::SecretTargetAttachment",
         "Properties":{
            "SecretId":{
               "Ref":"MyDocDBClusterRotationSecret"
            },
            "TargetId":{
               "Ref":"MyDocDBCluster"
            },
            "TargetType":"AWS::DocDB::DBCluster"
         }
      },
      "MySecretRotationSchedule":{
         "Type":"AWS::SecretsManager::RotationSchedule",
         "DependsOn":"SecretDocDBClusterAttachment",
         "Properties":{
            "SecretId":{
               "Ref":"MyDocDBClusterRotationSecret"
            },
            "HostedRotationLambda":{
               "RotationType":"MongoDBSingleUser",
               "RotationLambdaName":"MongoDBSingleUser",
               "VpcSecurityGroupIds":{
                  "Fn::GetAtt":[
                     "TestVPC",
                     "DefaultSecurityGroup"
                  ]
               },
               "VpcSubnetIds":{
                  "Fn::Join":[
                     ",",
                     [
                        {
                           "Ref":"TestSubnet01"
```

```
                        },
                        {
                           "Ref":"TestSubnet02"
                        }
                     ]
                  ]
               }
            },
            "RotationRules":{
              "Duration": "2h",
              "ScheduleExpression": "cron(0 8 1 * ? *)"
            }
         }
      }
   }
}
```

YAML

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::SecretsManager-2020-07-23
Resources:
  TestVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
      EnableDnsHostnames: true
      EnableDnsSupport: true
  TestSubnet01:
    Type: AWS::EC2::Subnet
    Properties:
      CidrBlock: 10.0.96.0/19
      AvailabilityZone: !Select
        - '0'
        - !GetAZs
          Ref: AWS::Region
      VpcId: !Ref TestVPC
  TestSubnet02:
    Type: AWS::EC2::Subnet
    Properties:
      CidrBlock: 10.0.128.0/19
      AvailabilityZone: !Select
        - '1'
```

```
        - !GetAZs
          Ref: AWS::Region
      VpcId: !Ref TestVPC
  SecretsManagerVPCEndpoint:
    Type: AWS::EC2::VPCEndpoint
    Properties:
      SubnetIds:
        - !Ref TestSubnet01
        - !Ref TestSubnet02
      SecurityGroupIds:
        - !GetAtt TestVPC.DefaultSecurityGroup
      VpcEndpointType: Interface
      ServiceName: !Sub com.amazonaws.${AWS::Region}.secretsmanager
      PrivateDnsEnabled: true
      VpcId: !Ref TestVPC
  MyDocDBClusterRotationSecret:
    Type: AWS::SecretsManager::Secret
    Properties:
      GenerateSecretString:
        SecretStringTemplate: '{"username": "someadmin","ssl": true}'
        GenerateStringKey: password
        PasswordLength: 16
        ExcludeCharacters: '"@/\'
      Tags:
        - Key: AppName
          Value: MyApp
  MyDocDBCluster:
    Type: AWS::DocDB::DBCluster
    Properties:
      DBSubnetGroupName: !Ref MyDBSubnetGroup
      MasterUsername: !Sub '{{resolve:secretsmanager:
${MyDocDBClusterRotationSecret}::username}}'
      MasterUserPassword: !Sub '{{resolve:secretsmanager:
${MyDocDBClusterRotationSecret}::password}}'
      VpcSecurityGroupIds:
        - !GetAtt TestVPC.DefaultSecurityGroup
  DocDBInstance:
    Type: AWS::DocDB::DBInstance
    Properties:
      DBClusterIdentifier: !Ref MyDocDBCluster
      DBInstanceClass: db.r5.large
  MyDBSubnetGroup:
    Type: AWS::DocDB::DBSubnetGroup
    Properties:
```

```
      DBSubnetGroupDescription: ''
      SubnetIds:
        - !Ref TestSubnet01
        - !Ref TestSubnet02
  SecretDocDBClusterAttachment:
    Type: AWS::SecretsManager::SecretTargetAttachment
    Properties:
      SecretId: !Ref MyDocDBClusterRotationSecret
      TargetId: !Ref MyDocDBCluster
      TargetType: AWS::DocDB::DBCluster
  MySecretRotationSchedule:
    Type: AWS::SecretsManager::RotationSchedule
    DependsOn: SecretDocDBClusterAttachment
    Properties:
      SecretId: !Ref MyDocDBClusterRotationSecret
      HostedRotationLambda:
        RotationType: MongoDBSingleUser
        RotationLambdaName: MongoDBSingleUser
        VpcSecurityGroupIds: !GetAtt TestVPC.DefaultSecurityGroup
        VpcSubnetIds: !Join
          - ','
          - - !Ref TestSubnet01
            - !Ref TestSubnet02
      RotationRules:
        Duration: 2h
        ScheduleExpression: cron(0 8 1 * ? *)
```

How Secrets Manager uses AWS CloudFormation

When you use the console to turn on rotation, Secrets Manager uses AWS CloudFormation to create resources for rotation. If you create a new rotation function during that process, CloudFormation creates an `AWS::Serverless::Function` based on the appropriate Rotation function templates. Then CloudFormation sets the `RotationSchedule`, which sets the rotation function and rotation rules for the secret. You can view the CloudFormation stack by choosing **View stack** in the banner after you turn on automatic rotation.

For information about turning on automatic rotation, see *Rotate secrets*.

Create AWS Secrets Manager secrets in AWS Cloud Development Kit (AWS CDK)

To create, manage, and retrieve secrets in a CDK app, you can use the AWS Secrets Manager Construct Library, which contains `ResourcePolicy`, `RotationSchedule`, `Secret`, `SecretRotation`, and `SecretTargetAttachment` constructs.

A good practice for using secrets in CDK applications is to first create the secret by using console or the CLI, and then import the secret into your CDK application.

For examples, see:

- Create a secret
- Import a secret
- Retrieve a secret
- Grant permission to use the secret
- Rotate a secret
- Rotate a database secret
- Replicate a secret to other Regions

For more information about the CDK, see the AWS Cloud Development Kit (AWS CDK) v2 Developer Guide.

Monitor AWS Secrets Manager secrets

AWS provides monitoring tools to watch Secrets Manager secrets, report when something is wrong, and take automatic actions when appropriate. You can use the logs if you need to investigate any unexpected usage or change, and then you can roll back unwanted changes. You can also set automated checks for inappropriate usage of secrets and any attempts to delete secrets.

Topics

Log AWS Secrets Manager events with AWS CloudTrail

AWS CloudTrail records all API calls for Secrets Manager as events, including calls from the Secrets Manager console, as well as several other events for rotation and secret version deletion. For a list of the log entries in Secrets Manager records, see CloudTrail entries.

You can use the CloudTrail console to view the last 90 days of recorded events. For an ongoing record of events in your AWS account, including events for Secrets Manager, create a trail so that CloudTrail delivers log files to an Amazon S3 bucket. See Creating a trail for your AWS account. You can also configure CloudTrail to receive CloudTrail log files from multiple AWS accounts and AWS Regions.

You can configure other AWS services to further analyze and act upon the data collected in CloudTrail logs. See AWS service integrations with CloudTrail logs. You can also get notifications when CloudTrail publishes new log files to your Amazon S3 bucket. See Configuring Amazon SNS notifications for CloudTrail.

To retrieve Secrets Manager events from CloudTrail logs (console)

1. Open the CloudTrail console at https://console.aws.amazon.com/cloudtrail/.

2. Ensure that the console points to the Region where your events occurred. The console shows only those events that occurred in the selected Region. Choose the Region from the drop-down list in the upper-right corner of the console.
3. In the left-hand navigation pane, choose **Event history**.
4. Choose **Filter** criteria and/or a **Time range** to help you find the event that you're looking for. For example:

   a. To see all Secrets Manager events, for **Lookup attributes**, choose **Event source**. Then, for **Enter event source**, choose **`secretsmanager.amazonaws.com`**.
   b. To see all events for a secret, for **Lookup attributes**, choose **Resource name**. Then, for **Enter a resource name**, enter the name of the secret.
5. To see additional details, choose the expand arrow next to the event. To see all of the information available, choose **View event**.

AWS CLI

Example Retrieve Secrets Manager events from CloudTrail logs

The following `lookup-events` example looks up Secrets Manager events.

```
aws cloudtrail lookup-events \
    --region us-east-1 \
    --lookup-attributes
 AttributeKey=EventSource,AttributeValue=secretsmanager.amazonaws.com
```

AWS CloudTrail entries for Secrets Manager

AWS Secrets Manager writes entries to your AWS CloudTrail log for all Secrets Manager operations and for other events related to rotation and deletion. For information about taking action on these events, see Match Secrets Manager events with EventBridge.

Log entry types

- Log entries for Secrets Manager operations
- Log entries for deletion
- Log entries for replication
- Log entries for rotation

Log entries for Secrets Manager operations

Events that are generated by calls to Secrets Manager operations have `"detail-type": ["AWS API Call via CloudTrail"]`.

> **Note**
>
> Before February 2024, some Secrets Manager operations reported events that contained "aRN" instead of "arn" for the secret ARN. For more information, see AWS re:Post.

The following are CloudTrail entries generated when you or a service call Secrets Manager operations through the API, SDK, or CLI.

BatchGetSecretValue

Generated by the BatchGetSecretValue operation. For information about retrieving secrets, see *Get secrets*.

CancelRotateSecret

Generated by the CancelRotateSecret operation. For information about rotation, see *Rotate secrets*.

CreateSecret

Generated by the CreateSecret operation. For information about creating secrets, see *Manage secrets*.

DeleteResourcePolicy

Generated by the DeleteResourcePolicy operation. For information about permissions, see the section called "Authentication and access control".

DeleteSecret

Generated by the DeleteSecret operation. For information about deleting secrets, see the section called "Delete a secret".

DescribeSecret

Generated by the DescribeSecret operation.

GetRandomPassword

Generated by the GetRandomPassword operation.

GetResourcePolicy

Generated by the GetResourcePolicy operation. For information about permissions, see the section called “Authentication and access control”.

GetSecretValue

Generated by the GetSecretValue and BatchGetSecretValue operations. For information about retrieving secrets, see *Get secrets*.

ListSecrets

Generated by the ListSecrets operation. For information about listing secrets, see the section called “Find secrets”.

ListSecretVersionIds

Generated by the ListSecretVersionIds operation.

PutResourcePolicy

Generated by the PutResourcePolicy operation. For information about permissions, see the section called “Authentication and access control”.

PutSecretValue

Generated by the PutSecretValue operation. For information about updating a secret, see the section called “Modify a secret”.

RemoveRegionsFromReplication

Generated by the RemoveRegionsFromReplication operation. For information about replicating a secret, see *Multi-region replication*.

ReplicateSecretToRegions

Generated by the ReplicateSecretToRegions operation. For information about replicating a secret, see *Multi-region replication*.

RestoreSecret

Generated by the RestoreSecret operation. For information about restoring a deleted secret, see the section called “Restore a secret”.

RotateSecret

Generated by the RotateSecret operation. For information about rotation, see *Rotate secrets*.

StopReplicationToReplica

Generated by the StopReplicationToReplica operation. For information about replicating a secret, see *Multi-region replication*.

TagResource

Generated by the TagResource operation. For information about tagging a secret, see the section called “Tag secrets”.

UntagResource

Generated by the UntagResource operation. For information about untagging a secret, see the section called “Tag secrets”.

UpdateSecret

Generated by the UpdateSecret operation. For information about updating a secret, see the section called “Modify a secret”.

UpdateSecretVersionStage

Generated by the UpdateSecretVersionStage operation. For information about version stages, see the section called “Secret versions”.

ValidateResourcePolicy

Generated by the ValidateResourcePolicy operation. For information about permissions, see the section called “Authentication and access control”.

Log entries for deletion

In addition to events for Secrets Manager operations, Secrets Manager generates the following events related to deletion. These events have `"detail-type": ["AWS Service Event via CloudTrail"]`.

CancelSecretVersionDelete

Generated by the Secrets Manager service. If you call `DeleteSecret` on a secret that has versions, and then later call `RestoreSecret`, Secrets Manager logs this event for each secret version that was restored. For information about restoring a deleted secret, see the section called “Restore a secret”.

EndSecretVersionDelete

Generated by the Secrets Manager service when a secret version is deleted. For more information, see the section called "Delete a secret".

StartSecretVersionDelete

Generated by the Secrets Manager service when Secrets Manager starts deletion for a secret version. For information about deleting secrets, see the section called "Delete a secret".

SecretVersionDeletion

Generated by the Secrets Manager service when Secrets Manager deletes a deprecated secret version. For more information, see Secret versions.

Log entries for replication

In addition to events for Secrets Manager operations, Secrets Manager generates the following events related to replication. These events have `"detail-type": ["AWS Service Event via CloudTrail"]`.

ReplicationFailed

Generated by the Secrets Manager service when replication fails. For information about replicating a secret, see *Multi-region replication*.

ReplicationStarted

Generated by the Secrets Manager service when Secrets Manager starts replicating a secret. For information about replicating a secret, see *Multi-region replication*.

ReplicationSucceeded

Generated by the Secrets Manager service when a secret is successfully replicated. For information about replicating a secret, see *Multi-region replication*.

Log entries for rotation

In addition to events for Secrets Manager operations, Secrets Manager generates the following events related to rotation. These events have `"detail-type": ["AWS Service Event via CloudTrail"]`.

RotationStarted

Generated by the Secrets Manager service when Secrets Manager starts rotating a secret. For information about rotation, see *Rotate secrets*.

RotationAbandoned

Generated by the Secrets Manager service when Secrets Manager abandons a rotation attempt and removes the `AWSPENDING` label from an existing version of a secret. Secrets Manager abandons rotation when you create a new version of a secret during rotation. For information about rotation, see *Rotate secrets*.

RotationFailed

Generated by the Secrets Manager service when rotation fails. For information about rotation, see the section called "Troubleshoot rotation".

RotationSucceeded

Generated by the Secrets Manager service when a secret is successfully rotated. For information about rotation, see *Rotate secrets*.

TestRotationStarted

Generated by the Secrets Manager service when Secrets Manager starts testing rotation for a secret that is not scheduled for immediate rotation. For information about rotation, see *Rotate secrets*.

TestRotationSucceeded

Generated by the Secrets Manager service when Secrets Manager successfully tests rotation for a secret that is not scheduled for immediate rotation. For information about rotation, see *Rotate secrets*.

TestRotationFailed

Generated by the Secrets Manager service when Secrets Manager tests rotation for a secret that is not scheduled for immediate rotation and rotation failed. For information about rotation, see the section called "Troubleshoot rotation".

Monitor AWS Secrets Manager with Amazon CloudWatch

Using Amazon CloudWatch, you can monitor AWS services and create alarms to let you know when metrics change. CloudWatch keeps these statistics for 15 months, so you can access historical information and gain a better perspective on how your web application or service is performing. For AWS Secrets Manager, you can monitor the number of secrets in your account, including secrets marked for deletion, and API calls to Secrets Manager, including calls made through the console. For information about how to monitor metrics, see Use CloudWatch metrics in the *CloudWatch User Guide*.

To find Secrets Manager metrics

1. On the CloudWatch console, under **Metrics**, choose **All metrics**.
2. In the **Metrics** search, box, enter `secret`.
3. Do the following:
   - To monitor the number of secrets in your account, choose **AWS/SecretsManager**, and then select **SecretCount**. This metric is published hourly.
   - To monitor API calls to Secrets Manager, including calls made through the console, choose **Usage > By AWS Resource**, and then select the API calls to monitor. For a list of Secrets Manager APIs, see Secrets Manager operations.
4. Do the following:
   - To create a graph of the metric, see Graphing metrics in the *Amazon CloudWatch User Guide*.
   - To detect anomalies, see Using CloudWatch anomaly detection in the *Amazon CloudWatch User Guide*.
   - To get statistics for a metric, see Get statistics for a metric in the *Amazon CloudWatch User Guide*.

CloudWatch alarms

You can create a CloudWatch alarm that sends an Amazon SNS message when the value of a metric changes and causes the alarm to change state. You can set an alarm on the Secrets Manager metric `ResourceCount`, which is the number of secrets in your account. You can also set alarms on An alarm watches a metric over a time period you specify, and performs actions based on the value of the metric relative to a given threshold over a number of time periods. Alarms invoke actions for sustained state changes only. CloudWatch alarms do not invoke actions simply because they are in a particular state; the state must have changed and been maintained for a specified number of periods.

For more information, see Using Amazon CloudWatch alarms and Create a CloudWatch alarm based on anomaly detection in the *CloudWatch User Guide*.

You can also set alarms that watch for certain thresholds, and send notifications or take actions when those thresholds are met. For more information, see the Amazon CloudWatch User Guide.

Match AWS Secrets Manager events with Amazon EventBridge

In Amazon EventBridge, you can match Secrets Manager events from CloudTrail log entries. You can configure EventBridge rules that look for these events and then send new generated events to a target to take action. For a list of CloudTrail entries that Secrets Manager logs, see CloudTrail entries. For instructions to set up EventBridge, see Getting started with EventBridge in the *EventBridge User Guide*.

Match all changes to a specified secret

> **Note**
>
> Because some Secrets Manager events return the ARN of the secret with different capitalization, in event patterns that match more than one action, to specify a secret by ARN, you may need to include both the keys `arn` and `aRN`. For more information, see AWS re:Post.

The following example shows an EventBridge event pattern that matches log entries for changes to a secret.

```
{
    "source": ["aws.secretsmanager"],
    "detail-type": ["AWS API Call via CloudTrail"],
    "detail": {
        "eventSource": ["secretsmanager.amazonaws.com"],
        "eventName": ["DeleteResourcePolicy", "PutResourcePolicy", "RotateSecret",
 "TagResource", "UntagResource", "UpdateSecret"],
        "responseElements": {
            "arn": ["arn:aws:secretsmanager:us-west-2:012345678901:secret:mySecret-
a1b2c3"]
        }
    }
}
```

Match events when a secret value rotates

The following example shows an EventBridge event pattern that matches CloudTrail log entries for secret value changes that occur from manual updates or automatic rotation. Because some of these events are from Secrets Manager operations and some are generated by the Secrets Manager service, you must include the `detail-type` for both.

```
{
    "source": ["aws.secretsmanager"],
    "$or": [
        { "detail-type": ["AWS API Call via CloudTrail"] },
        { "detail-type": ["AWS Service Event via CloudTrail"] }
    ],
    "detail": {
        "eventSource": ["secretsmanager.amazonaws.com"],
        "eventName": ["PutSecretValue", "UpdateSecret", "RotationSucceeded"]
    }
}
```

Monitor when AWS Secrets Manager secrets scheduled for deletion are accessed

You can use a combination of AWS CloudTrail, Amazon CloudWatch Logs, and Amazon Simple Notification Service (Amazon SNS) to create an alarm that notifies you of any attempts to access a secret pending deletion. If you receive a notification from an alarm, you might want to cancel deletion of the secret to give yourself more time to determine if you really want to delete it. Your investigation might result in the secret being restored because you still need the secret. Alternatively, you might need to update the user with details of the new secret to use.

The following procedures explain how to receive a notification when a request for the `GetSecretValue` operation that results in a specific error message written to your CloudTrail log files. Other API operations can be performed on the secret without triggering the alarm. This CloudWatch alarm detects usage that might indicate a person or application using outdated credentials.

Before you begin these procedures, you must turn on CloudTrail in the AWS Region and account where you intend to monitor AWS Secrets Manager API requests. For instructions, go to Creating a trail for the first time in the *AWS CloudTrail User Guide*.

Step 1: Configure CloudTrail log file delivery to CloudWatch Logs

You must configure delivery of your CloudTrail log files to CloudWatch Logs. You do this so CloudWatch Logs can monitor them for Secrets Manager API requests to retrieve a secret pending deletion.

To configure CloudTrail log file delivery to CloudWatch Logs

1. Open the CloudTrail console at https://console.aws.amazon.com/cloudtrail/.
2. On the top navigation bar, choose the AWS Region to monitor secrets.
3. In the left navigation pane, choose **Trails**, and then choose the name of the trail to configure for CloudWatch.
4. On the **Trails Configuration** page, scroll down to the **CloudWatch Logs** section, and then choose the edit icon ().
5. For **New or existing log group**, type a name for the log group, such as **`CloudTrail/MyCloudWatchLogGroup`**.
6. For **IAM role**, you can use the default role named **CloudTrail_CloudWatchLogs_Role**. This role has a default role policy with the required permissions to deliver CloudTrail events to the log group.
7. Choose **Continue** to save your configuration.
8. On the **AWS CloudTrail will deliver CloudTrail events associated with API activity in your account to your CloudWatch Logs log group** page, choose **Allow**.

Step 2: Create the CloudWatch alarm

To receive a notification when a Secrets Manager `GetSecretValue` API operation requests to access a secret pending deletion, you must create a CloudWatch alarm and configure notification.

To create a CloudWatch alarm

1. Sign in to the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. On the top navigation bar, choose the AWS Region where you want to monitor secrets.
3. In the left navigation pane, choose **Logs**.

4. In the list of **Log Groups**, select the check box next to the log group you created in the previous procedure, such as **CloudTrail/MyCloudWatchLogGroup**. Then choose **Create Metric Filter**.
5. For **Filter Pattern**, type or paste the following:

```
{ $.eventName = "GetSecretValue" && $.errorMessage = "*secret because it was marked
 for deletion*" }
```

   Choose **Assign Metric**.
6. On the **Create Metric Filter and Assign a Metric** page, do the following:
   a. For **Metric Namespace**, type **`CloudTrailLogMetrics`**.
   b. For **Metric Name**, type **`AttemptsToAccessDeletedSecrets`**.
   c. Choose **Show advanced metric settings**, and then if necessary for **Metric Value**, type **1**.
   d. Choose **Create Filter**.
7. In the filter box, choose **Create Alarm**.
8. In the **Create Alarm** window, do the following:
   a. For **Name**, type **`AttemptsToAccessDeletedSecretsAlarm`**.
   b. **Whenever:**, for **is:**, choose **>=**, and then type **1**.
   c. Next to **Send notification to:**, do one of the following:
      - To create and use a new Amazon SNS topic, choose **New list**, and then type a new topic name. For **Email list:**, type at least one email address. You can type more than one email address by separating them with commas.
      - To use an existing Amazon SNS topic, choose the name of the topic to use. If a list doesn't exist, choose **Select list**.
   d. Choose **Create Alarm**.

Step 3: Test the CloudWatch alarm

To test your alarm, create a secret and then schedule it for deletion. Then, try to retrieve the secret value. You shortly receive an email at the address you configured in the alarm. It alerts you to the use of a secret scheduled for deletion.

Monitor AWS Secrets Manager secrets for compliance by using AWS Config

You can use AWS Config to evaluate your secrets to see if they are in compliance with your standards. You define your internal security and compliance requirements for secrets using AWS Config rules. Then AWS Config can identify secrets that don't conform to your rules. You can also track changes to secret metadata, rotation configuration, the KMS key used for secret encryption, the Lambda rotation function, and tags associated with a secret.

You can configure AWS Config to notify you of changes. For more information, see Notifications that AWS Config sends to an Amazon SNS topic.

If you have secrets in multiple AWS accounts and AWS Regions in your organization, you can aggregate that configuration and compliance data. For more information, see Multi-account Multi-Region data aggregation.

To assess whether secrets are in compliance

- Follow the instructions on Evaluating your resources with AWS Config rules, and choose one of the following rules:
  - `secretsmanager-secret-unused`— Checks whether secrets were accessed within the specified number of days.
  - `secretsmanager-using-cmk` — Checks whether secrets are encrypted using the AWS managed key `aws/secretsmanager` or a customer managed key you created in AWS KMS.
  - `secretsmanager-rotation-enabled-check` — Checks whether rotation is configured for secrets stored in Secrets Manager.
  - `secretsmanager-scheduled-rotation-success-check`— Checks whether the last successful rotation is within the configured rotation frequency. The minimum frequency for the check is daily.
  - `secretsmanager-secret-periodic-rotation`— Checks whether secrets were rotated within the specified number of days.

Monitor Secrets Manager costs

You can use Amazon CloudWatch to monitor estimated AWS Secrets Manager charges. For more information, see Creating a billing alarm to monitor your estimated AWS charges in the *CloudWatch User Guide*.

Another option for monitoring your costs is AWS Cost Anomaly Detection. For more information, see Detecting unusual spend with AWS Cost Anomaly Detection in the *AWS Cost Management User Guide*.

For information about monitoring your Secrets Manager usage, see the section called "Monitor with CloudWatch" and the section called "Log with AWS CloudTrail ".

For information about AWS Secrets Manager pricing, see the section called "Pricing".

Detect threats with Amazon GuardDuty

Amazon GuardDuty is a threat detection service that helps you protect your accounts, containers, workloads, and the data with your AWS environment. By using machine learning (ML) models and anomaly and threat detection capabilities, GuardDuty continuously monitors different log sources to identify and prioritize potential security risks and malicious activities in your environment. For example, GuardDuty will detect potential threats such as unusual or suspicious access to secrets, and credential exfiltration in case it detects credentials that were created exclusively for an Amazon EC2 instance through an instance launch role but are being used from another account within AWS. For more information, see the Amazon GuardDuty User Guide.

Another example use-case for detection is anomalous behavior. For example, if AWS Secrets Manager typically gets `create-secret`, `get-secret-value`, `describe-secret`, and `list-secrets` calls from an entity using the Java SDK, and then a different entity begins calling `batch-get-secret-value` and `get-secret-value` using the AWS CLI from outside of the VPN, GuardDuty can report a finding that the second entity is anomalously invoking APIs. For more information, see GuardDuty IAM finding type CredentialAccess:IAMUser/AnomalousBehavior.

Compliance validation for AWS Secrets Manager

Your compliance responsibility when using Secrets Manager is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- *AWS Config* assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations. For more information, see the section called “Monitor secrets for compliance”.
- AWS Security Hub CSPM provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices. For information about using Security Hub CSPM to evaluate Secrets Manager resources, see AWS Secrets Manager controls in the *AWS Security Hub CSPM User Guide*.
- *IAM Access Analyzer* analyzes policies, including condition statements in a policy, that allow an external entity to access a secret. For more information, see Previewing access with Access Analyzer.
- *AWS Systems Manager* provides predefined runbooks for Secrets Manager. For more information, see Systems Manager Automation runbook reference for Secrets Manager.
- You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Compliance standards

AWS Secrets Manager has undergone auditing for the following standards and can be part of your solution when you need to obtain compliance certification.

- **HIPAA** – AWS has expanded its Health Insurance Portability and Accountability Act (HIPAA) compliance program to include AWS Secrets Manager as a HIPAA-eligible service. If you have an executed Business Associate Agreement (BAA) with AWS, you can use Secrets Manager to help build your HIPAA-compliant applications. AWS offers a HIPAA-focused whitepaper for customers who are interested in learning more about how they can leverage AWS for the processing and storage of health information. For more information, see HIPAA Compliance.

- **PCI Participating Organization** – AWS Secrets Manager has an Attestation of Compliance for Payment Card Industry (PCI) Data Security Standard (DSS) version 3.2 at Service Provider Level 1. Customers who use AWS products and services to store, process, or transmit cardholder data can use AWS Secrets Manager as they manage their own PCI DSS compliance certification. For more information about PCI DSS, including how to request a copy of the AWS PCI Compliance Package, see PCI DSS Level 1.
- **ISO** – AWS Secrets Manager has successfully completed compliance certification for ISO/IEC 27001, ISO/IEC 27017, ISO/IEC 27018, and ISO 9001. For more information, see ISO 27001, ISO 27017, ISO 27018, ISO 9001.
- **AICPA SOC** – System and Organization Control (SOC) reports are independent third-party examination reports that demonstrate how Secrets Manager achieves key compliance controls and objectives. The purpose of these reports is to help you and your auditors understand the AWS controls that are established to support operations and compliance. For more information, see SOC Compliance.
- **FedRAMP** – The Federal Risk and Authorization Management Program (FedRAMP) is a government-wide program that provides a standardized approach to security assessment, authorization, and continuous monitoring for cloud products and services. The FedRAMP Program also provides provisional authorizations for services and regions for East/West and GovCloud to consume government or regulated data. For more information, see FedRAMP Compliance.
- **Department of Defense** – The Department of Defense (DoD) Cloud Computing Security Requirements Guide (SRG) provides a standardized assessment and authorization process for cloud service providers (CSPs) to gain a DoD provisional authorization, so that they can serve DoD customers. For more information, see DoD SRG Resources
- **IRAP** – The Information Security Registered Assessors Program (IRAP) enables Australian government customers to validate that appropriate controls are in place and determine the appropriate responsibility model for addressing the requirements of the Australian government Information Security Manual (ISM) produced by the Australian Cyber Security Centre (ACSC). For more information, see IRAP Resources
- **OSPAR** – Amazon Web Services (AWS) achieved the Outsourced Service Provider's Audit Report (OSPAR) attestation. AWS alignment with the Association of Banks in Singapore (ABS) Guidelines

on Control Objectives and Procedures for Outsourced Service Providers (ABS Guidelines) demonstrates to customers AWS commitment to meeting the high expectations for cloud service providers set by the financial services industry in Singapore. For more information, see OSPAR Resources

Security in AWS Secrets Manager

Security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture built to meet the requirements of the most security-sensitive organizations.

You and AWS share the responsibility for security. The shared responsibility model describes this as security of the cloud and security in the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to AWS Secrets Manager, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your AWS service determines your responsibility. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

For more resources, see Security Pillar – AWS Well-Architected Framework.

Topics

Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets

When you use the AWS Command Line Interface (AWS CLI) to invoke AWS operations, you enter those commands in a command shell. For example, you can use the Windows command prompt or Windows PowerShell, or the Bash or Z shell, among others. Many of these command shells include functionality designed to increase productivity. But this functionality can be used to compromise your secrets. For example, in most shells, you can use the up arrow key to see the last entered command. The *command history* feature can be exploited by anyone who accesses your unsecured session. Also, other utilities that work in the background might have access to your command parameters, with the intended goal of helping you perform your tasks more efficiently. To mitigate such risks, ensure you take the following steps:

- Always lock your computer when you walk away from your console.
- Uninstall or disable console utilities you don't need or no longer use.
- Ensure the shell or the remote access program, if you are using one or the other, don't log typed commands.
- Use techniques to pass parameters not captured by the shell command history. The following example shows how you can type the secret text into a text file, and then pass the file to the AWS Secrets Manager command and immediately destroy the file. This means the typical shell history doesn't capture the secret text.

  The following example shows typical Linux commands but your shell might require slightly different commands:

```
$ touch secret.txt
        # Creates an empty text file
$ chmod go-rx secret.txt
        # Restricts access to the file to only the user
$ cat > secret.txt
        # Redirects standard input (STDIN) to the text file
ThisIsMyTopSecretPassword^D
        # Everything the user types from this point up to the CTRL-D (^D) is saved in
 the file
$ aws secretsmanager create-secret --name TestSecret --secret-string file://
secret.txt       # The Secrets Manager command takes the --secret-string parameter
 from the contents of the file
```

```
$ shred -u secret.txt
        # The file is destroyed so it can no longer be accessed.
```

After you run these commands, you should be able to use the up and down arrows to scroll through the command history and see that the secret text isn't displayed on any line.

> **Important**
>
> By default, you can't perform an equivalent technique in Windows unless you first reduce the size of the command history buffer to **1**.

To configure the Windows Command Prompt to have only 1 command history buffer of 1 command

1. Open an Administrator command prompt (**Run as administrator**).
2. Choose the icon in the upper left and then choose **Properties**.
3. On the **Options** tab, set **Buffer Size** and **Number of Buffers** both to **1**, and then choose **OK**.
4. Whenever you have to type a command you don't want in the history, immediately follow it with one other command, such as:

   ```
   echo.
   ```

   This ensures you flush the sensitive command.

For the Windows Command Prompt shell, you can download the [SysInternals SDelete](https://docs.microsoft.com/en-us/sysinternals/downloads/sdelete) tool, and then use commands similar to the following:

```
C:\> echo. 2> secret.txt
        # Creates an empty file
C:\> icacls secret.txt /remove "BUILTIN\Administrators" "NT AUTHORITY/SYSTEM" /
inheritance:r   # Restricts access to the file to only the owner
C:\> copy con secret.txt /y
        # Redirects the keyboard to text file, suppressing prompt to overwrite
THIS IS MY TOP SECRET PASSWORD^Z
      # Everything the user types from this point up to the CTRL-Z (^Z) is saved in the
 file
```

```
C:\> aws secretsmanager create-secret --name TestSecret --secret-string file://
secret.txt      # The Secrets Manager command takes the --secret-string parameter from
 the contents of the file
C:\> sdelete secret.txt
        # The file is destroyed so it can no longer be accessed.
```

Authentication and access control for AWS Secrets Manager

Secrets Manager uses AWS Identity and Access Management (IAM) to secure access to secrets. IAM provides authentication and access control. *Authentication* verifies the identity of individuals' requests. Secrets Manager uses a sign-in process with passwords, access keys, and multi-factor authentication (MFA) tokens to verify the identity of the users. See Signing in to AWS. *Access control* ensures that only approved individuals can perform operations on AWS resources such as secrets. Secrets Manager uses policies to define who has access to which resources, and which actions the identity can take on those resources. See Policies and permissions in IAM.

Topics

Permissions reference for AWS Secrets Manager

The permissions reference for Secrets Manager is available at Actions, resources, and condition keys for AWS Secrets Manager in the *Service Authorization Reference*.

Secrets Manager administrator permissions

To grant Secrets Manager administrator permissions, follow the instructions at Adding and removing IAM identity permissions, and attach the following policies:

- SecretsManagerReadWrite
- IAMFullAccess

We recommend you do not grant administrator permissions to end users. While this allows your users to create and manage their secrets, the permission required to enable rotation (IAMFullAccess) grants significant permissions that are not appropriate for end users.

Permissions to access secrets

By using IAM permission policies, you control which users or services have access to your secrets. A *permissions policy* describes who can perform which actions on which resources. You can:

- the section called “Identity-based policies”
- the section called “Resource-based policies”

Permissions for Lambda rotation functions

Secrets Manager uses AWS Lambda functions to rotate secrets. The Lambda function must have access to the secret as well as the database or service that the secret contains credentials for. See Permissions for rotation.

Permissions for encryption keys

Secrets Manager uses AWS Key Management Service (AWS KMS) keys to encrypt secrets. The AWS managed key `aws/secretsmanager` automatically has the correct permissions. If you use a different KMS key, Secrets Manager needs permissions to that key. See the section called “Permissions for the KMS key”.

Permissions for replication

By using IAM permission policies, you control which users or services can replicate your secrets to other Regions. See the section called “Prevent replication”.

Identity-based policies

You can attach permissions policies to IAM identities: users, user groups, and roles. In an identity-based policy, you specify which secrets the identity can access and the actions the identity can perform on the secrets. For more information, see Adding and removing IAM identity permissions.

You can grant permissions to a role that represents an application or user in another service. For example, an application running on an Amazon EC2 instance might need access to a database. You can create an IAM role attached to the EC2 instance profile and then use a permissions policy to grant the role access to the secret that contains credentials for the database. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances. Other services that you can attach roles to include Amazon Redshift, AWS Lambda, and Amazon ECS.

You can also grant permissions to users authenticated by an identity system other than IAM. For example, you can associate IAM roles to mobile app users who sign in with Amazon Cognito. The role grants the app temporary credentials with the permissions in the role permission policy. Then you can use a permissions policy to grant the role access to the secret. For more information, see Identity providers and federation.

You can use identity-based policies to:

- Grant an identity access to multiple secrets.
- Control who can create new secrets, and who can access secrets that haven't been created yet.
- Grant an IAM group access to secrets.

Examples:

- Example: Permission to retrieve individual secret values
- Example: Permission to read and describe individual secrets
- Example: Permission to retrieve a group of secret values in a batch
- Example: Wildcards
- Example: Permission to create secrets
- Example: Deny a specific AWS KMS key to encrypt secrets

Example: Permission to retrieve individual secret values

To grant permission to retrieve secret values, you can attach policies to secrets or identities. For help determining which type of policy to use, see Identity-based policies and resource-based policies. For information about how to attach a policy, see the section called "Resource-based policies" and the section called "Identity-based policies".

This example is useful when you want to grant access to an IAM group. To grant permission to retrieve a group of secrets in a batch API call, see the section called "Example: Permission to retrieve a group of secret values in a batch".

Example Read a secret that is encrypted using a customer managed key

If a secret is encrypted using a customer managed key, you can grant access to read the secret by attaching the following policy to an identity. \

JSON

```
{
"Version":"2012-10-17",
"Statement": [
{
  "Effect": "Allow",
  "Action": "secretsmanager:GetSecretValue",
  "Resource": "arn:aws:secretsmanager:us-east-1:123456789012:secret:secretName-AbCdEf"
},
{
  "Effect": "Allow",
  "Action": "kms:Decrypt",
  "Resource": "arn:aws:kms:us-east-1:123456789012:key/key-id"
}
]
}
```

Example: Permission to read and describe individual secrets

Example Read and describe one secret

You can grant access to a secret by attaching the following policy to an identity.

JSON

```
{
"Version":"2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "secretsmanager:GetSecretValue",
      "secretsmanager:DescribeSecret"
      ],
      "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName-AbCdEf"
    }
]
}
```

Example: Permission to retrieve a group of secret values in a batch

Example Read a group of secrets in a batch

You can grant access to retrieve a group of secrets in a batch API call by attaching the following policy to an identity. The policy restricts the caller so that they can only retrieve the secrets specified by *SecretARN1*, *SecretARN2*, and *SecretARN3*, even if the batch call includes other secrets. If the caller also requests other secrets in the batch API call, Secrets Manager won't return them. For more information, see `BatchGetSecretValue`..

JSON

```
{
"Version":"2012-10-17",
"Statement": [
{
  "Effect": "Allow",
  "Action": [
    "secretsmanager:BatchGetSecretValue",
    "secretsmanager:ListSecrets"
  ],
  "Resource": "*"
},
```

```
{
  "Effect": "Allow",
  "Action": [
    "secretsmanager:GetSecretValue"
  ],
  "Resource": [
    "arn:aws:secretsmanager:us-east-1:123456789012:secret:secretName1-AbCdEf",
    "arn:aws:secretsmanager:us-east-1:123456789012:secret:secretName2-AbCdEf",
    "arn:aws:secretsmanager:us-east-1:123456789012:secret:secretName3-AbCdEf"
    ]
}
]
}
```

Example: Wildcards

You can use wildcards to include a set of values in a policy element.

Example Access all secrets in a path

The following policy grants access to retrieve all secrets with a name beginning with "*TestEnv/*".

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "secretsmanager:GetSecretValue",
        "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:TestEnv/*"
    }
}
```

Example Access metadata on all secrets

The following policy grants `DescribeSecret` and permissions beginning with `List`: `ListSecrets` and `ListSecretVersionIds`.

JSON

```
{
"Version":"2012-10-17",
"Statement": {
"Effect": "Allow",
"Action": [
  "secretsmanager:DescribeSecret",
  "secretsmanager:List*"
],
"Resource": "*"
}
}
```

Example Match secret name

The following policy grants all Secrets Manager permissions for a secret by name. To use this policy, see the section called “Identity-based policies”.

To match a secret name, you create the ARN for the secret by putting together the Region, Account ID, secret name, and the wildcard (?) to match individual random characters. Secrets Manager appends six random characters to secret names as part of their ARN, so you can use this wildcard to match those characters. If you use the syntax `"another_secret_name-*"`, Secrets Manager matches not only the intended secret with the 6 random characters, but also matches `"another_secret_name-<anything-here>a1b2c3"`.

Because you can predict all of the parts of the ARN of a secret except the 6 random characters, using the wildcard character `'??????'` syntax enables you to securely grant permissions to a secret that doesn't yet exist. Be aware, however, if you delete the secret and recreate it with the same name, the user automatically receives permission to the new secret, even though the 6 characters changed.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": "secretsmanager:*",
            "Resource": [
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:a_specific_secret_name-a1b2c3",
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:another_secret_name-??????"
            ]
        }
    ]
}
```

Example: Permission to create secrets

To grant a user permissions to create a secret, we recommend you attach a permissions policy to an IAM group the user belongs to. See IAM user groups.

Example Create secrets

The following policy grants permission to create secrets and view a list of secrets. To use this policy, see the section called "Identity-based policies".

JSON

```
{
"Version":"2012-10-17",
"Statement": [
{
  "Effect": "Allow",
  "Action": [
    "secretsmanager:CreateSecret",
    "secretsmanager:ListSecrets"
  ],
  "Resource": "*"
}
]
}
```

Example: Deny a specific AWS KMS key to encrypt secrets

> **Important**
>
> To deny a customer managed key, we recommend you restrict access using a key policy or key grant. For more information, see [Authentication and access control for AWS KMS]() in the *AWS Key Management Service Developer Guide*.

Example Deny the AWS managed key `aws/secretsmanager`

The following policy denies the use of the AWS managed key `aws/secretsmanager` for creating or updating secrets. This policy requires secrets to be encrypted using a customer managed key. The policy includes two statements:

1. The first statement, `Sid: "RequireCustomerManagedKeysOnSecrets"`, denies requests for creating or updating secrets using the AWS managed key `aws/secretsmanager`.
2. The second statement, `Sid: "RequireKmsKeyIdParameterOnCreate"`, denies requests for creating secrets that don't include a KMS key, because Secrets Manager would default to using the AWS managed key `aws/secretsmanager`.

JSON

```
{
"Version":"2012-10-17",
"Statement": [
  {
      "Sid": "RequireCustomerManagedKeysOnSecrets",
      "Effect": "Deny",
      "Action": [
          "secretsmanager:CreateSecret",
          "secretsmanager:UpdateSecret"
      ],
      "Resource": "*",
      "Condition": {
          "StringLikeIfExists": {
              "secretsmanager:KmsKeyArn": "<key_ARN_of_the_AWS_managed_key>"
          }
      }
```

```
  },
  {
      "Sid": "RequireKmsKeyIdParameterOnCreate",
      "Effect": "Deny",
      "Action": "secretsmanager:CreateSecret",
      "Resource": "*",
      "Condition": {
          "Null": {
              "secretsmanager:KmsKeyArn": "true"
          }
      }
  }
]
}
```

Resource-based policies

In a resource-based policy, you specify who can access the secret and the actions they can perform on the secret. You can use resource-based policies to:

- Grant access to a single secret to multiple users and roles.
- Grant access to users or roles in other AWS accounts.

When you attach a resource-based policy to a secret in the console, Secrets Manager uses the automated reasoning engine [Zelkova](https://aws.amazon.com/blogs/security/protect-sensitive-data-in-the-cloud-with-automated-reasoning-zelkova/) and the API `ValidateResourcePolicy` to prevent you from granting a wide range of IAM principals access to your secrets. Alternatively, you can call the `PutResourcePolicy` API with the `BlockPublicPolicy` parameter from the CLI or SDK.

> **Important**
>
> Resource policy validation and the `BlockPublicPolicy` parameter help protect your resources by preventing public access from being granted through the resource policies that are directly attached to your secrets. In addition to using these features, carefully inspect the following policies to confirm that they do not grant public access:
>
> - Identity-based policies attached to associated AWS principals (for example, IAM roles)
> - Resource-based policies attached to associated AWS resources (for example, AWS Key Management Service (AWS KMS) keys)

To review permissions to your secrets, see Determine who has permissions to your secrets.

To view, change, or delete the resource policy for a secret (console)

1. Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/.
2. From the list of secrets, choose your secret.
3. On the secret details page, on the **Overview** tab, in the **Resource permissions** section, choose **Edit permissions**.
4. In the code field, do one of the following, and then choose **Save**:
   - To attach or modify a resource policy, enter the policy.
   - To delete the policy, clear the code field.

AWS CLI

Example Retrieve a resource policy

The following `get-resource-policy` example retrieves the resource-based policy attached to a secret.

```
aws secretsmanager get-resource-policy \
    --secret-id MyTestSecret
```

Example Delete a resource policy

The following `delete-resource-policy` example deletes the resource-based policy attached to a secret.

```
aws secretsmanager delete-resource-policy \
    --secret-id MyTestSecret
```

Example Add a resource policy

The following `put-resource-policy` example adds a permissions policy to a secret, checking first that the policy does not provide broad access to the secret. The policy is read from a file. For more information, see Loading AWS CLI parameters from a file in the AWS CLI User Guide.

```
aws secretsmanager put-resource-policy \
    --secret-id MyTestSecret \
    --resource-policy file://mypolicy.json \
    --block-public-policy
```

Contents of `mypolicy.json`:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/MyRole"
            },
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "*"
        }
    ]
}
```

AWS SDK

To retrieve the policy attached to a secret, use `GetResourcePolicy`.

To delete a policy attached to a secret, use `DeleteResourcePolicy`.

To attach a policy to a secret, use `PutResourcePolicy`. If there is already a policy attached, the command replaces it with the new policy. The policy must be formatted as JSON structured text. See JSON policy document structure.

For more information, see the section called “AWS SDKs”.

Examples

Examples:

- Example: Permission to retrieve individual secret values
- Example: Permissions and VPCs

- Example: Service principal

Example: Permission to retrieve individual secret values

To grant permission to retrieve secret values, you can attach policies to secrets or identities. For help determining which type of policy to use, see Identity-based policies and resource-based policies. For information about how to attach a policy, see the section called "Resource-based policies" and the section called "Identity-based policies".

This example is useful when you want to grant access to a single secret to multiple users or roles. To grant permission to retrieve a group of secrets in a batch API call, see the section called "Example: Permission to retrieve a group of secret values in a batch".

Example Read one secret

You can grant access to a secret by attaching the following policy to the secret.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/EC2RoleToAccessSecrets"
            },
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "*"
        }
    ]
}
```

Example: Permissions and VPCs

If you need to access Secrets Manager from within a VPC, you can make sure that requests to Secrets Manager come from the VPC by including a condition in your permissions policies. For more information, see Limit requests with VPC endpoint conditions and the section called "VPC endpoints (AWS PrivateLink)".

Make sure that requests to access the secret from other AWS services also come from the VPC, otherwise this policy will deny them access.

Example Require requests to come through a VPC endpoint

The following policy allows a user to perform Secrets Manager operations only when the request comes through the VPC endpoint *vpce-1234a5678b9012c*.

JSON

```
{
"Id": "example-policy-1",
"Version":"2012-10-17",
"Statement": [
{
  "Sid": "RestrictGetSecretValueoperation",
  "Effect": "Deny",
  "Principal": "*",
  "Action": "secretsmanager:GetSecretValue",
  "Resource": "*",
  "Condition": {
    "StringNotEquals": {
      "aws:sourceVpce": "vpce-12345678"
    }
  }
}
]
}
```

Example Require requests to come from a VPC

The following policy allows commands to create and manage secrets only when they come from *vpc-12345678*. In addition, the policy allows operations that use access the secret encrypted value only when the requests come from `vpc-2b2b2b2b`. You might use a policy like this one if you run an application in one VPC, but you use a second, isolated VPC for management functions.

JSON

```
{
"Id": "example-policy-2",
```

```
"Version":"2012-10-17",
"Statement": [
{
  "Sid": "AllowAdministrativeActionsfromONLYvpc-12345678",
  "Effect": "Deny",
  "Principal": "*",
  "Action": [
    "secretsmanager:Create*",
    "secretsmanager:Put*",
    "secretsmanager:Update*",
    "secretsmanager:Delete*",
    "secretsmanager:Restore*",
    "secretsmanager:RotateSecret",
    "secretsmanager:CancelRotate*",
    "secretsmanager:TagResource",
    "secretsmanager:UntagResource"
  ],
  "Resource": "*",
  "Condition": {
    "StringNotEquals": {
      "aws:sourceVpc": "vpc-12345678"
    }
  }
},
{
  "Sid": "AllowSecretValueAccessfromONLYvpc-2b2b2b2b",
  "Effect": "Deny",
  "Principal": "*",
  "Action": [
    "secretsmanager:GetSecretValue"
  ],
  "Resource": "*",
  "Condition": {
    "StringNotEquals": {
      "aws:sourceVpc": "vpc-2b2b2b2b"
    }
  }
}
]
}
```

Example: Service principal

If the resource policy attached to your secret includes an AWS service principal, we recommend that you use the aws:SourceArn and aws:SourceAccount global condition keys. The ARN and account values are included in the authorization context only when a request comes to Secrets Manager from another AWS service. This combination of conditions avoids a potential confused deputy scenario.

If a resource ARN includes characters that are not permitted in a resource policy, you cannot use that resource ARN in the value of the `aws:SourceArn` condition key. Instead, use the `aws:SourceAccount` condition key. For more information, see IAM requirements.

Service principals are not typically used as principals in a policy attached to a secret, but some AWS services require it. For information about resource policies that a service requires you to attach to a secret, see the service's documentation.

Example Allow a service to access a secret using a service principal

JSON

```
{
"Version":"2012-10-17",
"Statement": [
{
  "Effect": "Allow",
  "Principal": {
    "Service": [
      "s3.amazonaws.com"
    ]
  },
  "Action": "secretsmanager:GetSecretValue",
  "Resource": "*",
  "Condition": {
    "ArnLike": {
      "aws:sourceArn": "arn:aws:s3::123456789012:*"
    },
    "StringEquals": {
      "aws:sourceAccount": "123456789012"
    }
  }
```

```
}
]
}
```

Control access to secrets using attribute-based access control (ABAC)

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes or characteristics of the user, the data, or the environment, such as the department, business unit, or other factors that could affect the authorization outcome. In AWS, these attributes are called *tags*.

Using tags to control permissions is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome. ABAC rules are evaluated dynamically at runtime, which means that the users' access to applications and data and the type of allowed operations automatically change based on the contextual factors in the policy. For example, if a user changes department, access is automatically adjusted without the need to update permissions or request new roles. For more information, see: What is ABAC for AWS?, Define permissions to access secrets based on tags., and Scale your authorization needs for Secrets Manager using ABAC with IAM Identity Center.

Example: Allow an identity access to secrets that have specific tags

The following policy allows `DescribeSecret` access on secrets with a tag with the key *`ServerName`* and the value *`ServerABC`*. If you attach this policy to an identity, the identity has permission to any secrets with that tag in the account.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": {
  "Effect": "Allow",
  "Action": "secretsmanager:DescribeSecret",
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "secretsmanager:ResourceTag/ServerName": "ServerABC"
      }
    }
```

```
  }
}
```

Example: Allow access only to identities with tags that match secrets' tags

The following policy allows any identities in the account `GetSecretValue` access to any secrets in the account where the identity's *`AccessProject`* tag has the same value as the secret's *`AccessProject`* tag.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": {
  "Effect": "Allow",
  "Principal": {
    "AWS": "123456789012"
  },
  "Condition": {
    "StringEquals": {
      "aws:ResourceTag/AccessProject": "${ aws:PrincipalTag/AccessProject }"
    }
  },
  "Action": "secretsmanager:GetSecretValue",
  "Resource": "*"
  }
}
```

AWS managed policy for AWS Secrets Manager

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining [customer managed policies] that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

AWS managed policy: SecretsManagerReadWrite

This policy provides read/write access to AWS Secrets Manager, including permission to describe Amazon RDS, Amazon Redshift, and Amazon DocumentDB resources, and permission to use AWS KMS to encrypt and decrypt secrets. This policy also provides permission to create AWS CloudFormation change sets, get rotation templates from an Amazon S3 bucket that is managed by AWS, list AWS Lambda functions, and describe Amazon EC2 VPCs. These permissions are required by the console to set up rotation with existing rotation functions.

To create new rotation functions, you must also have permission to create AWS CloudFormation stacks and AWS Lambda execution roles. You can assign the IAMFullAccess managed policy. See Permissions for rotation.

Permissions details

This policy includes the following permissions.

- `secretsmanager` – Allows principals to perform all Secrets Manager actions.
- `cloudformation` – Allows principals to create CloudFormation stacks. This is required so that principals using the console to turn on rotation can create Lambda rotation functions through CloudFormation stacks. For more information, see the section called "How Secrets Manager uses CloudFormation".
- `ec2` – Allows principals to describe Amazon EC2 VPCs. This is required so that principals using the console can create rotation functions in the same VPC as the database of the credentials they are storing in a secret.
- `kms` – Allows principals to use AWS KMS keys for cryptographic operations. This is required so that Secrets Manager can encrypt and decrypt secrets. For more information, see the section called "Secret encryption and decryption".

- `lambda` – Allows principals to list Lambda rotation functions. This is required so that principals using the console can choose existing rotation functions.
- `rds` – Allows principals to describe clusters and instances in Amazon RDS. This is required so that principals using the console can choose Amazon RDS clusters or instances.
- `redshift` – Allows principals to describe clusters in Amazon Redshift. This is required so that principals using the console can choose Amazon Redshift clusters.
- `redshift-serverless` – Allows principals to describe namespaces in Amazon Redshift Serverless. This is required so that principals using the console can choose Amazon Redshift Serverless namespaces.
- `docdb-elastic` – Allows principals to describe elastic clusters in Amazon DocumentDB. This is required so that principals using the console can choose Amazon DocumentDB elastic clusters.
- `tag` – Allows principals to get all resources in the account that are tagged.
- `serverlessrepo` – Allows principals to create CloudFormation change sets. This is required so that principals using the console can create Lambda rotation functions. For more information, see the section called "How Secrets Manager uses CloudFormation".
- `s3` – Allows principals to get objects from an Amazon S3 bucket that is managed by AWS. This bucket contains Lambda Rotation function templates. This permission is required so that principals using the console can create Lambda rotation functions based on the templates in the bucket. For more information, see the section called "How Secrets Manager uses CloudFormation".

To view the policy, see SecretsManagerReadWrite JSON policy document.

AWS managed policy: AWSSecretsManagerClientReadOnlyAccess

This policy provides read-only access to AWS Secrets Manager secrets for client applications. It allows principals to retrieve secret values and describe secret metadata, along with the necessary AWS KMS permissions to decrypt secrets that are encrypted with customer-managed keys.

Permissions details

This policy includes the following permissions.

- `secretsmanager` – Allows principals to retrieve secret values and describe secret metadata.
- `kms` – Allows principals to decrypt secrets using AWS KMS keys. This permission is scoped to keys used by Secrets Manager through service-specific conditions.

To view more details about the policy, including the latest version of the JSON policy document, see AWSSecretsManagerClientReadOnlyAccess in the *AWS Managed Policy Reference Guide*.

Secrets Manager updates to AWS managed policies

View details about updates to AWS managed policies for Secrets Manager.

Change	Description	Date	Version
AWSSecretsManagerClientReadOnlyAccess – New managed policy	Secrets Manager created a new managed policy to provide read-only access to secrets for client applications. This policy allows retrieving secret values and describing secret metadata, with the necessary AWS KMS permissions to decrypt secrets.	November 5, 2025	v1
SecretsManagerReadWrite – Update to an existing policy	This policy was updated to allow describe access to Amazon Redshift Serverless so that console users can choose a Amazon Redshift Serverless namespace when they create an Amazon Redshift secret.	March 12, 2024	v5

Change	Description	Date	Version
SecretsManagerReadWrite – Update to an existing policy	This policy was updated to allow describe access to Amazon DocumentDB elastic clusters so that console users can choose an elastic cluster when they create an Amazon DocumentDB secret.	September 12, 2023	v4
SecretsManagerReadWrite – Update to an existing policy	This policy was updated to allow describe access to Amazon Redshift so that console users can choose a Amazon Redshift cluster when they create an Amazon Redshift secret. The update also added new permissions to allow read access to an Amazon S3 bucket managed by AWS that stores the Lambda rotation function templates.	June 24, 2020	v3

Change	Description	Date	Version
SecretsManagerReadWrite – Update to an existing policy	This policy was updated to allow describe access to Amazon RDS clusters so that console users can choose a cluster when they create an Amazon RDS secret.	May 3, 2018	v2
SecretsManagerReadWrite – New policy	Secrets Manager created a policy to grant permissions that are needed for using the console with all read/write access to Secrets Manager.	April 04, 2018	v1

Determine who has permissions to your AWS Secrets Manager secrets

By default, IAM identities don't have permission to access secrets. When authorizing access to a secret, Secrets Manager evaluates the resource-based policy attached to the secret and all identity-based policies attached to the IAM user or role sending the request. To do this, Secrets Manager uses a process similar to the one described in Determining whether a request is allowed or denied in the *IAM User Guide*.

When multiple policies apply to a request, Secrets Manager uses a hierarchy to control permissions:

1. If a statement in any policy with an explicit `deny` matches the request action and resource:

   The explicit `deny` overrides everything else and blocks the action.

2. If there is no explicit `deny`, but a statement with an explicit `allow` matches the request action and resource:

   The explicit `allow` grants the action in the request access to the resources in the statement.

If the identity and the secret are in two different accounts, there must be an `allow` in both the resource policy for the secret and the policy attached to the identity, otherwise AWS denies the request. For more information, see [Cross-account access](#).

3. If there is no statement with an explicit `allow` that matches the request action and resource:

   AWS denies the request by default, which is called an *implicit* deny.

To view the resource-based policy for a secret

- Do one of the following:
  - Open the Secrets Manager console at https://console.aws.amazon.com/secretsmanager/. In the secret details page for your secret, in the **Resource permissions** section, choose **Edit permissions**.
  - Use the AWS CLI to call `get-resource-policy` or AWS SDK to call `GetResourcePolicy`.

To determine who has access through identity-based policies

- Use the IAM policy simulator. See Testing IAM policies with the IAM policy simulator

Access AWS Secrets Manager secrets from a different account

To allow users in one account to access secrets in another account (*cross-account access*), you must allow access both in a resource policy and in an identity policy. This is different than granting access to identities in the same account as the secret.

Cross-account permission is effective only for the following operations:

- CancelRotateSecret
- DeleteResourcePolicy
- DeleteSecret
- DescribeSecret
- GetRandomPassword
- GetResourcePolicy
- GetSecretValue

- ListSecretVersionIds
- PutResourcePolicy
- PutSecretValue
- RemoveRegionsFromReplication
- ReplicateSecretToRegions
- RestoreSecret
- RotateSecret
- StopReplicationToReplica
- TagResource
- UntagResource
- UpdateSecret
- UpdateSecretVersionStage
- ValidateResourcePolicy

You can use the `BlockPublicPolicy` parameter with the PutResourcePolicy action to help protect your resources by preventing public access from being granted through the resource policies that are directly attached to your secrets. You can also use IAM Access Analyzer to verify cross-account access.

You must also allow the identity to use the KMS key that the secret is encrypted with. This is because you can't use the AWS managed key (`aws/secretsmanager`) for cross-account access. Instead, you must encrypt your secret with a KMS key that you create, and then attach a key policy to it. There is a charge for creating KMS keys. To change the encryption key for a secret, see the section called “Modify a secret”.

Important

Resource-based policies granting `secretsmanager:PutResourcePolicy` permission gives principals, even those in other accounts, the ability to modify your resource-based policies. This permission lets principals escalate existing permissions like obtaining full administrative access to secrets. We recommend you apply the principle of least privileged access to your policies. For more information, see Resource-based policies.

The following example policies assume you have a secret and encryption key in *Account1*, and an identity in *Account2* that you want to allow to access the secret value.

Step 1: Attach a resource policy to the secret in *Account1*

- The following policy allows `ApplicationRole` in `Account2` to access the secret in `Account1`. To use this policy, see the section called "Resource-based policies".

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/ApplicationRole"
            },
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "*"
        }
    ]
}
```

Step 2: Add a statement to the key policy for the KMS key in *Account1*

- The following key policy statement allows `ApplicationRole` in `Account2` to use the KMS key in `Account1` to decrypt the secret in `Account1`. To use this statement, add it to the key policy for your KMS key. For more information, see Changing a key policy.

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::Account2:role/ApplicationRole"
  },
  "Action": [
    "kms:Decrypt",
    "kms:DescribeKey"
  ],
  "Resource": "*"
```

```
}
```

Step 3: Attach an identity policy to the identity in *Account2*

- The following policy allows *ApplicationRole* in *Account2* to access the secret in *Account1* and decrypt the secret value by using the encryption key which is also in *Account1*. To use this policy, see the section called "Identity-based policies". You can find the ARN for your secret in the Secrets Manager console on the secret details page under **Secret ARN**. Alternatively, you can call `describe-secret`.

  JSON

  ```
  {
      "Version":"2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Action": "secretsmanager:GetSecretValue",
              "Resource": "arn:aws:secretsmanager:us-
  east-1:123456789012:secret:secretName-AbCdEf"
          },
          {
              "Effect": "Allow",
              "Action": "kms:Decrypt",
              "Resource": "arn:aws:kms:us-
  east-1:123456789012:key/EncryptionKey"
          }
      ]
  }
  ```

Access secrets from an on-premises environment

You can use AWS Identity and Access Management Roles Anywhere to obtain temporary security credentials in IAM for workloads such as servers, containers, and applications that run outside of AWS. Your workloads can use the same IAM policies and IAM roles that you use with AWS applications to access AWS resources. With IAM Roles Anywhere, you can use Secrets Manager to store and manage credentials that can be accessed by resources in AWS as well as on-premises devices such as application servers. For more information, see the IAM Roles Anywhere User Guide.

Data protection in AWS Secrets Manager

The AWS shared responsibility model applies to data protection in AWS Secrets Manager. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. Secrets Manager supports TLS 1.2 and 1.3 in all Regions. Secrets Manager also supports a hybrid post-quantum key exchange option for TLS (PQTLS) network encryption protocol.
- Sign your programmatic requests to Secrets Manager by using an access key ID and a secret access key associated with an IAM principal. Or you can use AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.
- Set up API and user activity logging with AWS CloudTrail. See the section called “Log with AWS CloudTrail ”.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. See the section called “Secrets Manager endpoints”.
- If you use the AWS CLI to access Secrets Manager, the section called “Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets”.

Encryption at rest

Secrets Manager uses encryption via AWS Key Management Service (AWS KMS) to protect the confidentiality of data at rest. AWS KMS provides a key storage and encryption service used by many AWS services. Every secret in Secrets Manager is encrypted with a unique data key. Each data key is protected by a KMS key. You can choose to use default encryption with the Secrets Manager

AWS managed key for the account, or you can create your own customer managed key in AWS KMS. Using a customer managed key gives you more granular authorization controls over your KMS key activities. For more information, see the section called "Secret encryption and decryption".

Encryption in transit

Secrets Manager provides secure and private endpoints for encrypting data in transit. The secure and private endpoints allows AWS to protect the integrity of API requests to Secrets Manager. AWS requires API calls be signed by the caller using X.509 certificates and/or a Secrets Manager Secret Access Key. This requirement is stated in the Signature Version 4 Signing Process (Sigv4).

If you use the AWS Command Line Interface (AWS CLI) or any of the AWS SDKs to make calls to AWS, you configure the access key to use. Then those tools automatically use the access key to sign the requests for you. See the section called "Mitigate the risks of using the AWS CLI to store your AWS Secrets Manager secrets".

Inter-network traffic privacy

AWS offers options for maintaining privacy when routing traffic through known and private network routes.

Traffic between service and on-premises clients and applications

You have two connectivity options between your private network and AWS Secrets Manager:

- An AWS Site-to-Site VPN connection. For more information, see What is AWS Site-to-Site VPN?
- An AWS Direct Connect connection. For more information, see What is AWS Direct Connect?

Traffic between AWS resources in the same Region

If you want to secure traffic between Secrets Manager and API clients in AWS, set up an AWS PrivateLink to privately access Secrets Manager API endpoints.

Encryption key management

When Secrets Manager needs to encrypt a new version of the protected secret data, Secrets Manager sends a request to AWS KMS to generate a new data key from the KMS key. Secrets Manager uses this data key for envelope encryption. Secrets Manager stores the encrypted data key with the encrypted secret. When the secret needs to be decrypted, Secrets Manager asks AWS KMS to decrypt the data key. Secrets Manager then uses the decrypted data key to decrypt the encrypted secret. Secrets Manager never stores the data key in unencrypted form and removes the key from memory as soon as possible. For more information, see the section called “Secret encryption and decryption”.

Secret encryption and decryption in AWS Secrets Manager

Secrets Manager uses envelope encryption with AWS KMS keys and data keys to protect each secret value. Whenever the secret value in a secret changes, Secrets Manager requests a new data key from AWS KMS to protect it. The data key is encrypted under a KMS key and stored in the metadata of the secret. To decrypt the secret, Secrets Manager first decrypts the encrypted data key using the KMS key in AWS KMS.

Secrets Manager does not use the KMS key to encrypt the secret value directly. Instead, it uses the KMS key to generate and encrypt a 256-bit Advanced Encryption Standard (AES) symmetric data key, and uses the data key to encrypt the secret value. Secrets Manager uses the plaintext data key to encrypt the secret value outside of AWS KMS, and then removes it from memory. It stores the encrypted copy of the data key in the metadata of the secret.

Topics

Choosing a AWS KMS key

When you create a secret, you can choose any symmetric encryption customer managed key in the AWS account and Region, or you can use the AWS managed key for Secrets Manager (`aws/`

secretsmanager). If you choose the AWS managed key `aws/secretsmanager` and it doesn't already exist yet, Secrets Manager creates it and associates it with the secret. You can use the same KMS key or different KMS keys for each secret in your account. You might want to use different KMS keys to set custom permissions on the keys for a group of secrets, or if you want to audit particular operations for those keys. Secrets Manager supports only [symmetric encryption KMS keys](). If you use a KMS key in an [external key store](), cryptographic operations on the KMS key might take longer and be less reliable and durable because the request has to travel outside of AWS.

For information about changing the encryption key for a secret, see [the section called “Change the encryption key for a secret”]().

When you change the encryption key, Secrets Manager re-encrypts `AWSCURRENT`, `AWSPENDING`, and `AWSPREVIOUS` versions with the new key. To avoid locking you out of the secret, Secrets Manager keeps all existing versions encrypted with the previous key. That means you can decrypt `AWSCURRENT`, `AWSPENDING`, and `AWSPREVIOUS` versions with the previous key or the new key. If you don't have `kms:Decrypt` permission to the previous key, when you change the encryption key, Secrets Manager can't decrypt the secret versions to re-encrypt them. In this case, the existing versions are not re-encrypted.

To make it so `AWSCURRENT` can only be decrypted by the new encryption key, create a new version of the secret with the new key. Then to be able to decrypt the `AWSCURRENT` secret version, you must have permission to the new key.

You can deny permission to the AWS managed key `aws/secretsmanager` and require secrets are encrypted with a customer managed key. For more information, see [the section called “Example: Deny a specific AWS KMS key to encrypt secrets”]().

To find the KMS key associated with a secret, view the secret in the console or call [ListSecrets]() or [DescribeSecret](). When the secret is associated with the AWS managed key for Secrets Manager (`aws/secretsmanager`), these operations do not return a KMS key identifier.

What is encrypted?

Secrets Manager encrypts the secret value, but it does not encrypt the following:

- Secret name and description
- Rotation settings
- ARN of the KMS key associated with the secret

- Any attached AWS tags

Encryption and decryption processes

To encrypt the secret value in a secret, Secrets Manager uses the following process.

1. Secrets Manager calls the AWS KMS GenerateDataKey operation with the ID of the KMS key for the secret and a request for a 256-bit AES symmetric key. AWS KMS returns a plaintext data key and a copy of that data key encrypted under the KMS key.
2. Secrets Manager uses the plaintext data key and the Advanced Encryption Standard (AES) algorithm to encrypt the secret value outside of AWS KMS. It removes the plaintext key from memory as soon as possible after using it.
3. Secrets Manager stores the encrypted data key in the metadata of the secret so it is available to decrypt the secret value. However, none of the Secrets Manager APIs return the encrypted secret or the encrypted data key.

To decrypt an encrypted secret value:

1. Secrets Manager calls the AWS KMS Decrypt operation and passes in the encrypted data key.
2. AWS KMS uses the KMS key for the secret to decrypt the data key. It returns the plaintext data key.
3. Secrets Manager uses the plaintext data key to decrypt the secret value. Then it removes the data key from memory as soon as possible.

Permissions for the KMS key

When Secrets Manager uses a KMS key in cryptographic operations, it acts on behalf of the user who is accessing or updating the secret value. You can grant permissions in an IAM policy or a key policy. The following Secrets Manager operations require AWS KMS permissions.

- CreateSecret
- GetSecretValue
- PutSecretValue
- UpdateSecret
- ReplicateSecretToRegions

To allow the KMS key to be used only for requests that originate in Secrets Manager, in the permissions policy, you can use the kms:ViaService condition key with the `secretsmanager.<Region>.amazonaws.com` value.

You can also use the keys or values in the encryption context as a condition for using the KMS key for cryptographic operations. For example, you can use a string condition operator in an IAM or key policy document, or use a grant constraint in a grant. KMS key grant propagation can take up to five minutes. For more information, see CreateGrant.

How Secrets Manager uses your KMS key

Secrets Manager calls the following AWS KMS operations with your KMS key.

GenerateDataKey

Secrets Manager calls the AWS KMS GenerateDataKey operation in response to the following Secrets Manager operations.

- CreateSecret – If the new secret includes a secret value, Secrets Manager requests a new data key to encrypt it.
- PutSecretValue – Secrets Manager requests a new data key to encrypt the specified secret value.
- ReplicateSecretToRegions – To encrypt the replicated secret, Secrets Manager requests a data key for the KMS key in the replica Region.
- UpdateSecret – If you change the secret value or the KMS key, Secrets Manager requests a new data key to encrypt the new secret value.

The RotateSecret operation does not call `GenerateDataKey`, because it does not change the secret value. However, if `RotateSecret` invokes a Lambda rotation function that changes the secret value, its call to the `PutSecretValue` operation triggers a `GenerateDataKey` request.

Decrypt

Secrets Manager calls the Decrypt operation in response to the following Secrets Manager operations.

- GetSecretValue and BatchGetSecretValue – Secrets Manager decrypts the secret value before returning it to the caller. To decrypt an encrypted secret value, Secrets Manager calls the AWS KMS Decrypt operation to decrypt the encrypted data key in the secret. Then, it uses the plaintext data key to decrypt the encrypted secret value. For batch commands, Secrets Manager can reuse the decrypted key, so not all calls result in a `Decrypt` request.

- PutSecretValue and UpdateSecret – Most `PutSecretValue` and `UpdateSecret` requests do not trigger a `Decrypt` operation. However, when a `PutSecretValue` or `UpdateSecret` request attempts to change the secret value in an existing version of a secret, Secrets Manager decrypts the existing secret value and compares it to the secret value in the request to confirm that they are the same. This action ensures the that Secrets Manager operations are idempotent. To decrypt an encrypted secret value, Secrets Manager calls the AWS KMS Decrypt operation to decrypt the encrypted data key in the secret. Then, it uses the plaintext data key to decrypt the encrypted secret value.
- ReplicateSecretToRegions – Secrets Manager first decrypts the secret value in the primary Region before re-encrypting the secret value with the KMS key in the replica Region.

Encrypt

Secrets Manager calls the Encrypt operation in response to the following Secrets Manager operations:

- UpdateSecret – If you change the KMS key, Secrets Manager re-encrypts the data key that protects the `AWSCURRENT`, `AWSPREVIOUS`, and `AWSPENDING` secret versions with the new key.

DescribeKey

Secrets Manager calls the DescribeKey operation to determine whether to list the KMS key when you create or edit a secret in the Secrets Manager console.

Validating access to the KMS key

When you establish or change the KMS key that is associated with secret, Secrets Manager calls the `GenerateDataKey` and `Decrypt` operations with the specified KMS key. These calls confirm that the caller has permission to use the KMS key for these operation. Secrets Manager discards the results of these operations; it does not use them in any cryptographic operation.

You can identify these validation calls because the value of the `SecretVersionId` key encryption context in these requests is `RequestToValidateKeyAccess`.

> **Note**
>
> In the past, Secrets Manager validation calls did not include an encryption context. You might find calls with no encryption context in older AWS CloudTrail logs.

Key policy of the AWS managed key (`aws/secretsmanager`)

The key policy for the AWS managed key for Secrets Manager (`aws/secretsmanager`) gives users permission to use the KMS key for specified operations only when Secrets Manager makes the request on the user's behalf. The key policy does not allow any user to use the KMS key directly.

This key policy, like the policies of all AWS managed keys, is established by the service. You cannot change the key policy, but you can view it at any time. For details, see Viewing a key policy.

The policy statements in the key policy have the following effect:

- Allow users in the account to use the KMS key for cryptographic operations only when the request comes from Secrets Manager on their behalf. The `kms:ViaService` condition key enforces this restriction.
- Allows the AWS account to create IAM policies that allow users to view KMS key properties and revoke grants.
- Although Secrets Manager does not use grants to gain access to the KMS key, the policy also allows Secrets Manager to create grants for the KMS key on the user's behalf and allows the account to revoke any grant that allows Secrets Manager to use the KMS key. These are standard elements of policy document for an AWS managed key.

The following is a key policy for an example AWS managed key for Secrets Manager.

JSON

```
{
  "Id": "auto-secretsmanager-2",
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "Allow access through AWS Secrets Manager for all principals in the
 account that are authorized to use AWS Secrets Manager",
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "*"
        ]
      },
      "Action": [
```

```
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:CreateGrant",
        "kms:DescribeKey"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "kms:CallerAccount": "111122223333",
          "kms:ViaService": "secretsmanager.us-west-2.amazonaws.com"
        }
      }
    },
    {
      "Sid": "Allow access through AWS Secrets Manager for all principals in the account that are authorized to use AWS Secrets Manager",
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "*"
        ]
      },
      "Action": "kms:GenerateDataKey*",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "kms:CallerAccount": "111122223333"
        },
        "StringLike": {
          "kms:ViaService": "secretsmanager.us-west-2.amazonaws.com"
        }
      }
    },
    {
      "Sid": "Allow direct access to key metadata to the account",
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::111122223333:root"
        ]
      },
      "Action": [
        "kms:Describe*",
```

```
                "kms:Get*",
                "kms:List*",
                "kms:RevokeGrant"
            ],
            "Resource": "*"
        }
    ]
}
```

Secrets Manager encryption context

An encryption context is a set of key–value pairs that contain arbitrary non-secret data. When you include an encryption context in a request to encrypt data, AWS KMS cryptographically binds the encryption context to the encrypted data. To decrypt the data, you must pass in the same encryption context.

In its GenerateDataKey and Decrypt requests to AWS KMS, Secrets Manager uses an encryption context with two name–value pairs that identify the secret and its version, as shown in the following example. The names do not vary, but combined encryption context values will be different for each secret value.

```
"encryptionContext": {
    "SecretARN": "arn:aws:secretsmanager:us-east-2:111122223333:secret:test-secret-a1b2c3",
    "SecretVersionId": "EXAMPLE1-90ab-cdef-fedc-ba987SECRET1"
}
```

You can use the encryption context to identify these cryptographic operation in audit records and logs, such as AWS CloudTrail and Amazon CloudWatch Logs, and as a condition for authorization in policies and grants.

The Secrets Manager encryption context consists of two name-value pairs.

- **SecretARN** – The first name–value pair identifies the secret. The key is `SecretARN`. The value is the Amazon Resource Name (ARN) of the secret.

  ```
  "SecretARN": "ARN of an Secrets Manager secret"
  ```

For example, if the ARN of the secret is `arn:aws:secretsmanager:us-east-2:111122223333:secret:test-secret-a1b2c3`, the encryption context would include the following pair.

```
"SecretARN": "arn:aws:secretsmanager:us-east-2:111122223333:secret:test-secret-a1b2c3"
```

- **SecretVersionId** – The second name–value pair identifies the version of the secret. The key is `SecretVersionId`. The value is the version ID.

```
"SecretVersionId": "<version-id>"
```

For example, if the version ID of the secret is `EXAMPLE1-90ab-cdef-fedc-ba987SECRET1`, the encryption context would include the following pair.

```
"SecretVersionId": "EXAMPLE1-90ab-cdef-fedc-ba987SECRET1"
```

When you establish or change the KMS key for a secret, Secrets Manager sends GenerateDataKey and Decrypt requests to AWS KMS to validate that the caller has permission to use the KMS key for these operations. It discards the responses; it does not use them on the secret value.

In these validation requests, the value of the `SecretARN` is the actual ARN of the secret, but the `SecretVersionId` value is `RequestToValidateKeyAccess`, as shown in the following example encryption context. This special value helps you to identify validation requests in logs and audit trails.

```
"encryptionContext": {
    "SecretARN": "arn:aws:secretsmanager:us-east-2:111122223333:secret:test-secret-a1b2c3",
    "SecretVersionId": "RequestToValidateKeyAccess"
}
```

Note

In the past, Secrets Manager validation requests did not include an encryption context. You might find calls with no encryption context in older AWS CloudTrail logs.

Monitor Secrets Manager interaction with AWS KMS

You can use AWS CloudTrail and Amazon CloudWatch Logs to track the requests that Secrets Manager sends to AWS KMS on your behalf. For information about monitoring the use of secrets, see *Monitor secrets*.

GenerateDataKey

When you create or change the secret value in a secret, Secrets Manager sends a GenerateDataKey request to AWS KMS that specifies the KMS key for the secret.

The event that records the `GenerateDataKey` operation is similar to the following example event. The request is invoked by `secretsmanager.amazonaws.com`. The parameters include the Amazon Resource Name (ARN) of the KMS key for the secret, a key specifier that requires a 256-bit key, and the encryption context that identifies the secret and version.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-05-31T23:23:41Z"
            }
        },
        "invokedBy": "secretsmanager.amazonaws.com"
    },
    "eventTime": "2018-05-31T23:23:41Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "secretsmanager.amazonaws.com",
    "userAgent": "secretsmanager.amazonaws.com",
    "requestParameters": {
        "keyId": "arn:aws:kms:us-
east-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        "keySpec": "AES_256",
```

```
        "encryptionContext": {
            "SecretARN": "arn:aws:secretsmanager:us-east-2:111122223333:secret:test-
secret-a1b2c3",
            "SecretVersionId": "EXAMPLE1-90ab-cdef-fedc-ba987SECRET1"
        }
    },
    "responseElements": null,
    "requestID": "a7d4dd6f-6529-11e8-9881-67744a270888",
    "eventID": "af7476b6-62d7-42c2-bc02-5ce86c21ed36",
    "readOnly": true,
    "resources": [
        {
            "ARN": "arn:aws:kms:us-
east-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
```

Decrypt

When you get or change the secret value of a secret, Secrets Manager sends a Decrypt request to AWS KMS to decrypt the encrypted data key. For batch commands, Secrets Manager can reuse the decrypted key, so not all calls result in a `Decrypt` request.

The event that records the `Decrypt` operation is similar to the following example event. The user is the principal in your AWS account who is accessing the table. The parameters include the encrypted table key (as a ciphertext blob) and the encryption context that identifies the table and the AWS account. AWS KMS derives the ID of the KMS key from the ciphertext.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
```

```
                "mfaAuthenticated": "false",
                "creationDate": "2018-05-31T23:36:09Z"
            }
        },
        "invokedBy": "secretsmanager.amazonaws.com"
    },
    "eventTime": "2018-05-31T23:36:09Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "secretsmanager.amazonaws.com",
    "userAgent": "secretsmanager.amazonaws.com",
    "requestParameters": {
        "encryptionContext": {
            "SecretARN": "arn:aws:secretsmanager:us-east-2:111122223333:secret:test-
secret-a1b2c3",
            "SecretVersionId": "EXAMPLE1-90ab-cdef-fedc-ba987SECRET1"
        }
    },
    "responseElements": null,
    "requestID": "658c6a08-652b-11e8-a6d4-ffee2046048a",
    "eventID": "f333ec5c-7fc1-46b1-b985-cbda13719611",
    "readOnly": true,
    "resources": [
        {
            "ARN": "arn:aws:kms:us-
east-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            "accountId": "111122223333",
            "type": "AWS::KMS::Key"
        }
    ],
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
}
```


Encrypt

When you change the KMS key associated with a secret, Secrets Manager sends an Encrypt request to AWS KMS to re-encrypt the AWSCURRENT, AWSPREVIOUS, and AWSPENDING secret versions with the new key. When you replicate a secret to another Region, Secrets Manager also sends an Encrypt request to AWS KMS.

The event that records the `Encrypt` operation is similar to the following example event. The user is the principal in your AWS account who is accessing the table.


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AROAIGDTESTANDEXAMPLE:user01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/user01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "creationDate": "2023-06-09T18:11:34Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "secretsmanager.amazonaws.com"
    },
    "eventTime": "2023-06-09T18:11:34Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Encrypt",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "secretsmanager.amazonaws.com",
    "userAgent": "secretsmanager.amazonaws.com",
    "requestParameters": {
        "keyId": "arn:aws:kms:us-east-2:111122223333:key/EXAMPLE1-f1c8-4dce-8777-
aa071ddefdcc",
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT",
        "encryptionContext": {
            "SecretARN": "arn:aws:secretsmanager:us-
east-2:111122223333:secret:ChangeKeyTest-5yKnKS",
            "SecretVersionId": "EXAMPLE1-5c55-4d7c-9277-1b79a5e8bc50"
        }
    },
    "responseElements": null,
    "requestID": "129bd54c-1975-4c00-9b03-f79f90e61d60",
    "eventID": "f7d9ff39-15ab-47d8-b94c-56586de4ab68",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
```

```
            "ARN": "arn:aws:kms:us-west-2:111122223333:key/EXAMPLE1-f1c8-4dce-8777-aa071ddefdcc"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```

Infrastructure security in AWS Secrets Manager

As a managed service, AWS Secrets Manager is protected by the AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well‐Architected Framework*.

Access to Secrets Manager via the network is through AWS published APIs using TLS. Secrets Manager APIs are callable from any network location. However, Secrets Manager supports resource-based access policies, which can include restrictions based on the source IP address. You can also use Secrets Manager resource policies to control access to secrets from specific virtual private cloud (VPC) endpoints, or specific VPCs. Effectively, this isolates network access to a given secret from only the specific VPC within the AWS network. For more information, see the section called “VPC endpoints (AWS PrivateLink)”.

Using an AWS Secrets Manager VPC endpoint

We recommend that you run as much of your infrastructure as possible on private networks that are not accessible from the public internet. You can establish a private connection between your VPC and Secrets Manager by creating an *interface VPC endpoint*. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to privately access Secrets Manager APIs without an internet gateway, NAT device, VPN connection, or Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with Secrets Manager APIs. Traffic between your VPC and Secrets Manager does not leave the AWS network. For more information, see Interface VPC endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*.

When Secrets Manager rotates a secret by using a Lambda rotation function, for example a secret that contains database credentials, the Lambda function makes requests to both the database and Secrets Manager. When you turn on automatic rotation by using the console, Secrets Manager creates the Lambda function in the same VPC as your database. We recommend that you create a Secrets Manager endpoint in the same VPC so that requests from the Lambda rotation function to Secrets Manager don't leave the Amazon network.

If you enable private DNS for the endpoint, you can make API requests to Secrets Manager using its default DNS name for the Region, for example, `secretsmanager.us-east-1.amazonaws.com`. For more information, see Accessing a service through an interface endpoint in the *Amazon VPC User Guide*.

You can make sure that requests to Secrets Manager come from the VPC access by including a condition in your permissions policies. For more information, see the section called “Example: Permissions and VPCs”.

You can use AWS CloudTrail logs to audit your use of secrets through the VPC endpoint.

To create a VPC endpoint for Secrets Manager

1. See Creating an interface endpoint in the *Amazon VPC User Guide*. Use one of the following service names:
   - `com.amazonaws.`*`region`*`.secretsmanager`
   - `com.amazonaws.`*`region`*`.secretsmanager-fips`
2. To control access to the endpoint, see Control access to VPC endpoints using endpoint policies.
3. To use IPv6 and dual-stack addressing, see IPv4 and IPv6 access.

Create an endpoint policy for your interface endpoint

An endpoint policy is an IAM resource that you can attach to an interface endpoint. The default endpoint policy allows full access to Secrets Manager through the interface endpoint. To control the access allowed to Secrets Manager from your VPC, attach a custom endpoint policy to the interface endpoint.

An endpoint policy specifies the following information:

- The principals that can perform actions (AWS accounts, IAM users, and IAM roles).
- The actions that can be performed.
- The resources on which the actions can be performed.

For more information, see Control access to services using endpoint policies in the *AWS PrivateLink Guide*.

Example: VPC endpoint policy for Secrets Manager actions

The following is an example of a custom endpoint policy. When you attach this policy to your interface endpoint, it grants access to the listed Secrets Manager actions on the specified secret.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "Allow all users to use GetSecretValue and DescribeSecret on the
 specified secret.",
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret"
      ],
      "Resource": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:secretName-AbCdEf"
    }
  ]
}
```

Shared subnets

You can't create, describe, modify, or delete VPC endpoints in subnets that are shared with you. However, you can use the VPC endpoints in subnets that are shared with you. For information about VPC sharing, see Share your VPC with other accounts in the *Amazon Virtual Private Cloud User Guide*.

Control API access with IAM policies

If you use IAM policies to control access to AWS services based on IP addresses, you might need to update your policies to include IPv6 address ranges. This guide explains the differences between IPv4 and IPv6 and describes how to update your IAM policies to support both protocols. Implementing these changes helps you maintain secure access to your AWS resources while supporting IPv6.

What is IPv6?

IPv6 is the next generation IP standard intended to eventually replace IPv4. The previous version, IPv4, uses a 32-bit addressing scheme to support 4.3 billion devices. IPv6 instead uses 128-bit addressing to support approximately 340 trillion trillion trillion (or 2 to the 128th power) devices.

For more information, see the VPC IPv6 web page.

These are examples of IPv6 addresses:

```
2001:cdba:0000:0000:0000:0000:3257:9652 # This is a full, unabbreviated IPv6 address.
2001:cdba:0:0:0:0:3257:9652             # The same address with leading zeros in each
 group omitted
2001:cdba::3257:965                     # A compressed version of the same address.
```

IAM dual-stack (IPv4 and IPv6) policies

You can use IAM policies to control access to Secrets Manager APIs and prevent IP addresses outside the configured range from accessing Secrets Manager APIs.

The *secretsmanager.{region}.amazonaws.com* dual-stack endpoint for Secrets Manager APIs supports both IPv6 and IPv4.

If you need to support both IPv4 and IPv6, update your IP address filtering policies to handle IPv6 addresses. Otherwise, you might not be able to connect to Secrets Manager over IPv6.

Who should make this change?

This change affects you if you use dual addressing with policies that contain `aws:sourceIp`. *Dual addressing* means that the network supports both IPv4 and IPv6.

If you use dual addressing, update your IAM policies that currently use IPv4 format addresses to include IPv6 format addresses.

Who should not make this change?

This change doesn't affect you if you *only* use IPv4 networks.

Adding IPv6 to an IAM policy

IAM policies use the `aws:SourceIp` condition key to control access from specific IP addresses. If your network uses dual addressing (IPv4 and IPv6), update your IAM policies to include IPv6 address ranges.

In the `Condition` element of your policies, use the `IpAddress` and `NotIpAddress` operators for IP address conditions. Don't use string operators, as they can't handle the various valid IPv6 address formats.

These examples use `aws:SourceIp`. For VPCs, use `aws:VpcSourceIp` instead.

The following is the Denies access to AWS based on the source IP reference policy from the *IAM User Guide*. The `NotIpAddress` in the `Condition` element to lists two IPv4 address ranges, `192.0.2.0/24` and `203.0.113.0/24`, which will be denied access to the API.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
        "Effect": "Deny",
        "Action": "*",
        "Resource": "*",
        "Condition": {
            "NotIpAddress": {
                "aws:SourceIp": [
                    "192.0.2.0/24",
                    "203.0.113.0/24"
                ]
            },
            "Bool": {
                "aws:ViaAWSService": "false"
            }
        }
    }
}
```

To update this policy, change the `Condition` element to include the IPv6 address ranges `2001:DB8:1234:5678::/64` and `2001:cdba:3257:8593::/64`.

> **Note**
>
> Don't remove the existing IPv4 addresses. They're needed for backward compatibility.

```
"Condition": {
                "NotIpAddress": {
                    "aws:SourceIp": [
                        "192.0.2.0/24", <<DO NOT REMOVE existing IPv4 address>>
                        "203.0.113.0/24", <<DO NOT REMOVE existing IPv4 address>>
                        "2001:DB8:1234:5678::/64", <<New IPv6 IP address>>
                        "2001:cdba:3257:8593::/64" <<New IPv6 IP address>>
                    ]
                },
                "Bool": {
                    "aws:ViaAWSService": "false"
                }
            }
```

To update this policy for a VPC, use `aws:VpcSourceIp` instead of `aws:SourceIp`:

```
"Condition": {
                "NotIpAddress": {
                    "aws:VpcSourceIp": [
                        "10.0.2.0/24", <<DO NOT REMOVE existing IPv4 address>>
                        "10.0.113.0/24", <<DO NOT REMOVE existing IPv4 address>>
                        "fc00:DB8:1234:5678::/64", <<New IPv6 IP address>>
                        "fc00:cdba:3257:8593::/64" <<New IPv6 IP address>>
                    ]
                },
                "Bool": {
                    "aws:ViaAWSService": "false"
                }
            }
```

Verifying your client supports IPv6

If you use the *secretsmanager.{region}.amazonaws.com* endpoint, verify that you can connect to it. The following steps describe how to perform the verification.

This examples uses Linux and curl version 8.6.0 and uses the AWS Secrets Manager service which has IPv6 enabled endpoints located at the **amazonaws.com** endpoint.

> **Note**
>
> The **secretsmanager.{region}.amazonaws.com** differs from the typical dual-stack naming convention. For a full list of Secrets Manager endpoints, see AWS Secrets Manager endpoints.
> Change the AWS Region to the same Region where your service is located. In this example, we use the US East (N. Virginia) – `us-east-1` endpoint.

1. Determine if the endpoint resolves with an IPv6 address using the following `dig` command.

```
$ dig +short AAAA secretsmanager.us-east-1.amazonaws.com

> 2600:1f18:e2f:4e05:1a8a:948e:7c08:c1c3
```

2. Determine if the client network can make an IPv6 connection using the following `curl` command. A 404 response code means the connection succeeded, while a 0 response code means the connection failed.

```
$ curl --ipv6 -o /dev/null --silent -w "\nremote ip: %{remote_ip}\nresponse code:
 %{response_code}\n" https://secretsmanager.us-east-1.amazonaws.com

> remote ip: 2600:1f18:e2f:4e05:1a8a:948e:7c08:c1c3
> response code: 404
```

If a remote IP was identified **and** the response code is not `0`, a network connection was successfully made to the endpoint using IPv6. The remote IP should be an IPv6 address because the operating system should select the protocol that is valid for the client.

If the remote IP is blank or the response code is `0`, the client network or the network path to the endpoint is IPv4-only. You can verify this configuration with the following `curl` command.

```
$ curl -o /dev/null --silent -w "\nremote ip: %{remote_ip}\nresponse code:
 %{response_code}\n" https://secretsmanager.us-east-1.amazonaws.com

> remote ip: 3.123.154.250
```

```
> response code: 404
```

Resiliency in AWS Secrets Manager

AWS builds the global infrastructure around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which connect with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones allow you to be more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information on resiliency and disaster recovery, refer to Reliability Pillar – AWS Well-Architected Framework.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Post-quantum TLS

Secrets Manager supports a hybrid post-quantum key exchange option for the Transport Layer Security (TLS) network encryption protocol. You can use this TLS option when you connect to Secrets Manager API endpoints. We're offering this feature before post-quantum algorithms are standardized so you can begin testing the effect of these key exchange protocols on Secrets Manager calls. These optional hybrid post-quantum key exchange features are at least as secure as the TLS encryption we use today and are likely to provide additional security benefits. However, they affect latency and throughput compared to the classic key exchange protocols in use today. The Secrets Manager Agent uses the post-quantum ML-KEM key exchange as the highest-priority key exchange by default.

To protect data encrypted today against potential future attacks, AWS is participating with the cryptographic community in the development of quantum-resistant or *post-quantum* algorithms. We've implemented hybrid post-quantum key exchange cipher suites in Secrets Manager endpoints. These hybrid cipher suites, which combine classic and post-quantum elements, ensure that your TLS connection is at least as strong as it would be with classic cipher suites. However, because the performance characteristics and bandwidth requirements of hybrid cipher suites are different from those of classic key exchange mechanisms, we recommend that you test them on your API calls.

Secrets Manager supports PQTLS in all Regions except China Regions.

To configure hybrid post-quantum TLS

1. Add the AWS Common Runtime client to your Maven dependencies. We recommend using the latest available version. For example, this statement adds version 2.20.0.

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>aws-crt-client</artifactId>
  <version>2.20.0</version>
</dependency>
```

2. Add the AWS SDK for Java 2.x to your project and initialize it. Enable the hybrid post-quantum cipher suites on your HTTP client.

```
SdkAsyncHttpClient awsCrtHttpClient = AwsCrtAsyncHttpClient.builder()
            .postQuantumTlsEnabled(true)
            .build();
```

3. Create the Secrets Manager asynchronous client.

```
SecretsManagerAsyncClient SecretsManagerAsync = SecretsManagerAsyncClient.builder()
            .httpClient(awsCrtHttpClient)
            .build();
```

   Now when you call Secrets Manager API operations, your calls are transmitted to the Secrets Manager endpoint using hybrid post-quantum TLS.

For more information about using hybrid post-quantum TLS, see:

- AWS SDK for Java 2.x Developer Guide and the AWS SDK for Java 2.x released blog post.
- Introducing s2n-tls, a New Open Source TLS Implementation and Using s2n-tls.
- Post-Quantum Cryptography at the National Institute for Standards and Technology (NIST).
- Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS).

Post-quantum TLS for Secrets Manager is available in all AWS Regions except China.

Troubleshooting AWS Secrets Manager

Use the information here to help you diagnose and fix issues that you might encounter when you're working with Secrets Manager.

For issues related to rotation, see the section called "Troubleshoot rotation".

Topics

"Access denied" messages

When you make an API call such as GetSecretValue or CreateSecret to Secrets Manager, you must have IAM permissions to make that call. When you use the console, the console makes the same API calls on your behalf, so you must also have IAM permissions. An administrator can grant permissions by attaching an IAM policy to your IAM user, or to a group that you're a member of. If the policy statements that grant those permissions include any conditions, such as time-of-day or IP address restrictions, you also must meet those requirements when you send the request. For information about viewing or modifying policies for an IAM user, group, or role, see Working with Policies in the *IAM User Guide*. For information about permissions required for Secrets Manager, see the section called "Authentication and access control".

If you're signing API requests manually, without using the AWS SDKs, verify you correctly signed the request.

"Access denied" for temporary security credentials

Verify the IAM user or role you're using to make the request has the correct permissions. Permissions for temporary security credentials derive from an IAM user or role. This means the permissions are limited to those granted to the IAM user or role. For more information about how permissions for temporary security credentials are determined, see Controlling Permissions for Temporary Security Credentials in the *IAM User Guide*.

Verify that your requests are signed correctly and that the request is well-formed. For details, see the toolkit documentation for your chosen SDK, or Using Temporary Security Credentials to Request Access to AWS Resources in the *IAM User Guide*.

Verify that your temporary security credentials haven't expired. For more information, see Requesting Temporary Security Credentials in the *IAM User Guide*.

For information about permissions required for Secrets Manager, see the section called “Authentication and access control”.

Changes I make aren't always immediately visible.

Secrets Manager uses a distributed computing model called eventual consistency. Any change that you make in Secrets Manager (or other AWS services) takes time to become visible from all possible endpoints. Some of the delay results from the time it takes to send the data from server to server, from replication zone to replication zone, and from region to region around the world. Secrets Manager also uses caching to improve performance, but in some cases this can add time. The change might not be visible until the previously cached data times out.

Design your global applications to account for these potential delays. Also, ensure that they work as expected, even when a change made in one location isn't instantly visible at another.

For more information about how some other AWS services are affected by eventual consistency, see:

- Managing data consistency in the *Amazon Redshift Database Developer Guide*
- Amazon S3 Data Consistency Model in the *Amazon Simple Storage Service User Guide*
- Ensuring Consistency When Using Amazon S3 and Amazon EMR for ETL Workflows in the AWS Big Data Blog
- Amazon EC2 Eventual Consistency in the *Amazon EC2 API Reference*

“Cannot generate a data key with an asymmetric KMS key” when creating a secret

Secrets Manager uses a symmetric encryption KMS key associated with a secret to generate a data key for each secret value. You can't use an asymmetric KMS key. Verify you are using a symmetric encryption KMS key instead of an asymmetric KMS key. For instructions, see Identifying asymmetric KMS keys.

An AWS CLI or AWS SDK operation can't find my secret from a partial ARN

In many cases, Secrets Manager can find your secret from part of an ARN rather than the full ARN. However, if your secret's name ends in a hyphen followed by six characters, Secrets Manager might not be able to find the secret from only part of an ARN. Instead, we recommend that you use the complete ARN or the name of the secret.

More details

Secrets Manager includes six random characters at the end of the secret name to help ensure that the secret ARN is unique. If the original secret is deleted, and then a new secret is created with the same name, the two secrets have different ARNs because of these characters. Users with access to the old secret don't automatically get access to the new secret because the ARNs are different.

Secrets Manager constructs an ARN for a secret with Region, account, secret name, and then a hyphen and six more characters, as follows:

```
arn:aws:secretsmanager:us-east-2:111122223333:secret:SecretName-abcdef
```

If your secret name ends with a hyphen and six characters, using only part of the ARN can appear to Secrets Manager as though you are specifying a full ARN. For example, you might have a secret named `MySecret-abcdef` with the ARN

`arn:aws:secretsmanager:us-east-2:111122223333:secret:MySecret-abcdef-nutBrk`

If you call the following operation, which only uses part of the secret ARN, then Secrets Manager might not find the secret.

```
$ aws secretsmanager describe-secret --secret-id arn:aws:secretsmanager:us-
east-2:111122223333:secret:MySecret-abcdef
```

This secret is managed by an AWS service, and you must use that service to update it.

If you encounter this message while trying to modify a secret, the secret can only be updated by using the managing service listed in the message. For more information, see *Secrets managed by other services*.

To determine who manages a secret, you can review the secret name. Secrets managed by other services are prefixed with the ID of that service. Or, in the AWS CLI, call describe-secret, and then review the field `OwningService`.

Python module import fails when using `Transform: AWS::SecretsManager-2024-09-16`

If you're using the Transform: `AWS::SecretsManager-2024-09-16` and encounter Python module import failures when your rotation Lambda function runs, the issue is likely caused by an incompatible `Runtime` value. With this transform version, AWS CloudFormation manages the runtime version, code, and shared object files for you. You don't need to manage these yourself.

AWS Secrets Manager quotas

Secrets Manager read APIs have high TPS quotas, and control plane APIs that are less frequently called have lower TPS quotas. We recommend you avoid calling `PutSecretValue` or `UpdateSecret` at a sustained rate of more than once every 10 minutes. When you call `PutSecretValue` or `UpdateSecret` to update the secret value, Secrets Manager creates a new version of the secret. Secrets Manager removes unlabeled versions when there are more than 100, but it does not remove versions created less than 24 hours ago. If you update the secret value more than once every 10 minutes, you create more versions than Secrets Manager removes, and you will reach the quota for secret versions.

You may operate multiple regions in your account, and each quota is specific to each region.

When an application in one AWS account uses a secret owned by a different account, it's known as a *cross-account request*. For cross-account requests, Secrets Manager throttles the account of the identity that makes the requests, not the account that owns the secret. For example, if an identity from account A uses a secret in account B, the secret use applies only to the quotas in account A.

Secrets Manager quotas

Name	Default	Adjus e	Description
Combined rate of DeleteResourcePoli cy, GetResourcePolicy, PutResour cePolicy, and ValidateResourcePolicy API requests	Each supported Region: 50 per second	No	The maximum transacti ons per second for DeleteResourcePoli cy, GetResourcePolicy, PutResourcePolicy, and ValidateResourcePolicy API requests combined.
Combined rate of PutSecretValue, RemoveRegionsFromReplication, ReplicateSecretToRegion, StopRepli cationToReplica, UpdateSecret, and UpdateSecretVersionStage API requests	Each supported Region: 50 per second	No	The maximum transacti ons per second for PutSecretValue, RemoveRegionsFromR eplication, Replicate SecretToRegion,

Name	Default	Adjustable	Description
			StopReplicationToReplica, UpdateSecret, and UpdateSecretVersionStage API requests combined.
Combined rate of RestoreSecret API requests	Each supported Region: 50 per second	No	The maximum transactions per second for RestoreSecret API requests.
Combined rate of RotateSecret and CancelRotateSecret API requests	Each supported Region: 50 per second	No	The maximum transactions per second for RotateSecret and CancelRotateSecret API requests combined.
Combined rate of TagResource and UntagResource API requests	Each supported Region: 50 per second	No	The maximum transactions per second for TagResource and UntagResource API requests combined.
Rate of BatchGetSecretValue API requests	Each supported Region: 100 per second	No	The maximum transactions per second for BatchGetSecretValue API requests.
Rate of CreateSecret API requests	Each supported Region: 50 per second	No	The maximum transactions per second for CreateSecret API requests.

Name	Default	Adjustable	Description
Rate of DeleteSecret API requests	Each supported Region: 50 per second	No	The maximum transactions per second for DeleteSecret API requests.
Rate of DescribeSecret API requests	Each supported Region: 40,000 per second	No	The maximum transactions per second for DescribeSecret API requests.
Rate of GetRandomPassword API requests	Each supported Region: 50 per second	No	The maximum transactions per second for GetRandomPassword API requests.
Rate of GetSecretValue API requests	Each supported Region: 10,000 per second	No	The maximum transactions per second for GetSecretValue API requests.
Rate of ListSecretVersionIds API requests	Each supported Region: 50 per second	No	The maximum transactions per second for ListSecretVersionIds API requests.
Rate of ListSecrets API requests	Each supported Region: 100 per second	No	The maximum transactions per second for ListSecrets API requests.
Resource-based policy length	Each supported Region: 20,480	No	The maximum number of characters in a resource-based permissions policy attached to a secret.

Name	Default	Adjustable	Description
Secret value size	Each supported Region: 65,536 Bytes	No	The maximum size of an encrypted secret value. If the secret value is a string, then this is the number of characters permitted in the secret value.
Secrets	Each supported Region: 500,000	No	The maximum number of secrets in each AWS Region of this AWS account.
Staging labels attached across all versions of a secret	Each supported Region: 20	No	The maximum number of staging labels attached across all versions of a secret.
Versions per secret	Each supported Region: 100	No	The maximum number of versions of a secret.

Add retries to your application

Your AWS client might see calls to Secrets Manager fail due to unexpected issues on the client side. Or calls might fail due to rate limiting from Secrets Manager. When you exceed an API request quota, Secrets Manager throttles the request. It rejects an otherwise valid request and returns a throttling error. For both kinds of failures, we recommend you retry the call after a brief waiting period. This is called a backoff and retry strategy.

If you experience the following errors, you might want to add retries to your application code:

Transient errors and exceptions

- `RequestTimeout`
- `RequestTimeoutException`

- `PriorRequestNotComplete`
- `ConnectionError`
- `HTTPClientError`

Service-side throttling and limit errors and exceptions

- `Throttling`
- `ThrottlingException`
- `ThrottledException`
- `RequestThrottledException`
- `TooManyRequestsException`
- `ProvisionedThroughputExceededException`
- `TransactionInProgressException`
- `RequestLimitExceeded`
- `BandwidthLimitExceeded`
- `LimitExceededException`
- `RequestThrottled`
- `SlowDown`

For more information, as well as example code, on retries, exponential backoff, and jitter, see the following resources:

- Exponential Backoff and Jitter
- Timeouts, retries and backoff with jitter
- Error retries and exponential backoff in AWS.

Document history

The following table describes the important changes to the documentation since the last release of AWS Secrets Manager. For notification about updates to this documentation, you can subscribe to an RSS feed.

Change	Description	Date
New AWS managed policy	Secrets Manager has released a new managed policy `AWSSecretsManagerClientReadOnlyAccess` that provides read-only access to secrets for client applications. For information, see Secrets Manager updates to AWS managed policies.	November 5, 2025
Added support for cost allocation tags	Secrets Manager now supports cost allocation tags, allowing customers to categorize and track costs by department, team, or application. For more information, see Using cost allocation tags with AWS Secrets Manager.	May 27, 2025
Added IPv6 and dual-stack support	Secrets Manager now supports dual-stack endpoints . See IPv4 and IPv6 access for more information.	December 20, 2024
Secrets Manager change to AWS managed policy	The `SecretsManagerReadWrite` managed policy now includes `redshift-serverless` permissio	March 12, 2024

	n. For more information, see [AWS managed policy for AWS Secrets Manager](#)	

Earlier updates

The following table describes important changes in each release of the AWS Secrets Manager User Guide before February 2024.

Change	Description	Date
General availability	This is the initial public release of Secrets Manager.	Apr 4, 2018