AWS Security Blog

Migrating your secrets to AWS Secrets Manager, Part 2: Implementation

by Adesh Gairola and Eric Swamy | on 21 JUL 2023 | in Advanced (300), AWS Secrets Manager, Best Practices, Security, Identity, & Compliance, Technical How-To | Permalink | Comments | Share

In Part 1 of this series, we provided guidance on how to discover and classify secrets and design a migration solution for customers who plan to migrate secrets to AWS Secrets Manager. We also mentioned steps that you can take to enable preventative and detective controls for Secrets Manager. In this post, we discuss how teams should approach the next phase, which is implementing the migration of secrets to Secrets Manager. We also provide a sample solution to demonstrate migration.

Implement secrets migration

Application teams lead the effort to design the migration strategy for their application secrets. Once you've made the decision to migrate your secrets to Secrets Manager, there are two potential options for migration implementation. One option is to move the application to AWS in its current state and then modify the application source code to retrieve secrets from Secrets Manager. Another option is to update the on-premises application to use Secrets Manager for retrieving secrets. You can use features such as AWS Identity and Access Management (IAM) Roles Anywhere to make the application communicate with Secrets Manager even before the migration, which can simplify the migration phase.

If the application code contains hardcoded secrets, the code should be updated so that it references Secrets Manager. A good interim state would be to pass these secrets as environment variables to your application. Using environment variables helps in decoupling the secrets retrieval logic from the application code and allows for a smooth cutover and rollback (if required).

Cutover to Secrets Manager should be done in a maintenance window. This minimizes downtime and impacts to production.

Before you perform the cutover procedure, verify the following:

- Application components can access Secrets Manager APIs. Based on your environment, this connectivity might be provisioned through interface virtual private cloud (VPC) endpoints or over the internet.
- Secrets exist in Secrets Manager and have the correct tags. This is important if you are using attribute-based access control (ABAC).
- Applications that integrate with Secrets Manager have the required IAM permissions.
- Have a well-documented cutover and rollback plan that contains the changes that will be made to the application during cutover. These would include steps like updating the code to use environment variables and updating the application to use IAM roles or instance profiles (for apps that are being migrated to Amazon Elastic Compute Cloud (Amazon EC2)).

After the cutover, verify that Secrets Manager integration was successful. You can use AWS CloudTrail to confirm that application components are using Secrets Manager.

We recommend that you further optimize your integration by enabling automatic secrets rotation. If your secrets were previously widely accessible (for example, they were stored in your Git repositories), we recommend rotating as soon as possible when migrating .

Sample application to demo integration with Secrets Manager

In the next sections, we present a sample AWS Cloud Development Kit (AWS CDK) solution that demonstrates the implementation of the previously discussed guardrails, design, and migration strategy. You can use the sample solution as a starting point and expand upon it. It includes components that environment teams may deploy to help provide potentially secure access for application teams to migrate their secrets to Secrets Manager. The solution uses ABAC, a tagging scheme, and IAM Roles Anywhere to demonstrate regulated access to secrets for application teams. Additionally, the solution contains client-side utilities to assist application and migration teams in updating secrets. Teams with on-premises applications that are seeking integration with Secrets Manager before migration can use the client-side utility for access through IAM Roles Anywhere.

The sample solution is hosted on the aws-secrets-manager-abac-authorization-samples GitHub repository and is made up of the following components:

- A common environment infrastructure stack (created and owned by environment teams). This stack provisions the following resources:
  - A sample VPC created with Amazon Virtual Private Cloud (Amazon VPC), with `PUBLIC`, `PRIVATE_WITH_NAT`, and `PRIVATE_ISOLATED` subnet types.
  - VPC endpoints for the AWS Key Management Service (AWS KMS) and Secrets Manager services to the sample VPC. The use of VPC endpoints means that calls to AWS KMS and Secrets Manager are not made over the internet and remain internal to the AWS backbone network.
  - An empty shell secret, tagged with the supplied attributes and an IAM managed policy that uses attribute-based access control conditions. This means that the secret is managed in code, but the actual secret value is not visible in version control systems like GitHub or in AWS CloudFormation parameter inputs.
- An IAM Roles Anywhere infrastructure stack (created and owned by environment teams). This stack provisions the following resources:
  - An AWS Certificate Manager Private Certificate Authority (AWS Private CA).
  - An IAM Roles Anywhere public key infrastructure (PKI) trust anchor that uses AWS Private CA.
  - An IAM role for the on-premises application that uses the common environment infrastructure stack.
  - An IAM Roles Anywhere profile.

    Note: You can choose to use your existing CAs as trust anchors. If you do not have a CA, the stack described here provisions a PKI for you. IAM Roles Anywhere allows migration teams to use Secrets Manager before the application is moved to the cloud. Post migration, you could consider updating the applications to use native IAM integration (like instance profiles for EC2 instances) and revoking IAM Roles Anywhere credentials.
- A client-side utility (primarily used by application or migration teams). This is a shell script that does the following:

  - Assists in provisioning a certificate by using OpenSSL.
  - Uses aws_signing_helper (Credential Helper) to set up AWS CLI profiles by using the `credential_process` for IAM Roles Anywhere.
  - Assists application teams to access and update their application secrets after assuming an IAM role by using IAM Roles Anywhere.

- A sample application stack (created and owned by the application/migration team). This is a sample serverless application that demonstrates the use of the solution. It deploys the following components, which indicate that your ABAC-based IAM strategy is working as expected and is effectively restricting access to secrets:
  - The sample application stack uses a VPC-deployed common environment infrastructure stack.
  - It deploys an Amazon Aurora MySQL serverless cluster in the `PRIVATE_ISOLATED` subnet and uses the secret that is created through a common environment infrastructure stack.
  - It deploys a sample Lambda function in the `PRIVATE_WITH_NAT` subnet.
  - It deploys two IAM roles for testing:
    - *allowedRole* (default role): When the application uses this role, it is able to use the `GET` action to get the secret and open a connection to the Aurora MySQL database.
    - *Not allowedRole*: When the application uses this role, it is unable to use the `GET` action to get the secret and open a connection to the Aurora MySQL database.

Prerequisites to deploy the sample solution

The following software packages need to be installed in your development environment before you deploy this solution:

- Node.js v12 or later (https://nodejs.org)
- AWS CLI version 2 (https://docs.aws.amazon.com/cli/latest/userguide/welcome-versions.html)
- jq (https://stedolan.github.io/jq/)
- Git (https://git-scm.com/)
- OpenSSL (https://www.openssl.org/)

> **Note:** In this section, we provide examples of AWS CLI commands and configuration for Linux or macOS operating systems. For instructions on using AWS CLI on Windows, refer to the AWS CLI documentation.

Before deployment, make sure that the correct AWS credentials are configured in your terminal session. The credentials can be either in the environment variables or in `~/.aws`. For more details, see Configuring the AWS CLI.

Next, use the following commands to set your AWS credentials to deploy the stack:

```
export AWS_ACCESS_KEY_ID=<>
export AWS_SECRET_ACCESS_KEY=<>
```

```
export AWS_REGION = <>
```

You can view the IAM credentials that are being used by your session by running the command `aws sts get-caller-identity`. If you are running the cdk command for the first time in your AWS account, you will need to run the following `cdk bootstrap` command to provision a CDK Toolkit stack that will manage the resources necessary to enable deployment of cloud applications with the AWS CDK.

```
cdk bootstrap aws://<AWS account number>/<Region> # Bootstrap CDK in the specified accour
```

Select the applicable archetype and deploy the solution

This section outlines the design and deployment steps for two archetypes:

- For customers with on-premises applications who want to make use of Secrets Manager, we provide the steps for integration. (See the section **Archetype 1: Application is currently on premises**.)
- For customers with applications already migrated to AWS, we demonstrate a sample integration of Secrets Manager with AWS Lambda. (See the section **Archetype 2: Application has migrated to AWS**.)

Archetype 1: Application is currently on premises

Archetype 1 has the following requirements:

- The application is currently hosted on premises.
- The application would consume API keys, stored credentials, and other secrets in Secrets Manager.

The application, environment and security teams work together to define a tagging strategy that will be used to restrict access to secrets. After this, the proposed workflow for each persona is as follows:

1. The *environment engineer* deploys a common environment infrastructure stack (as described earlier in this post) to bootstrap the AWS account with secrets and IAM policy by using the supplied tagging requirement.
2. Additionally, the *environment engineer* deploys the IAM Roles Anywhere infrastructure stack.
3. The *application developer* updates the secrets required by the application by using the client-side utility (`helper.sh`).
4. The *application developer* uses the client-side utility to update the AWS CLI profile to consume the IAM Roles Anywhere role from the on-premises servers.

   Figure 1 shows the workflow for Archetype 1.

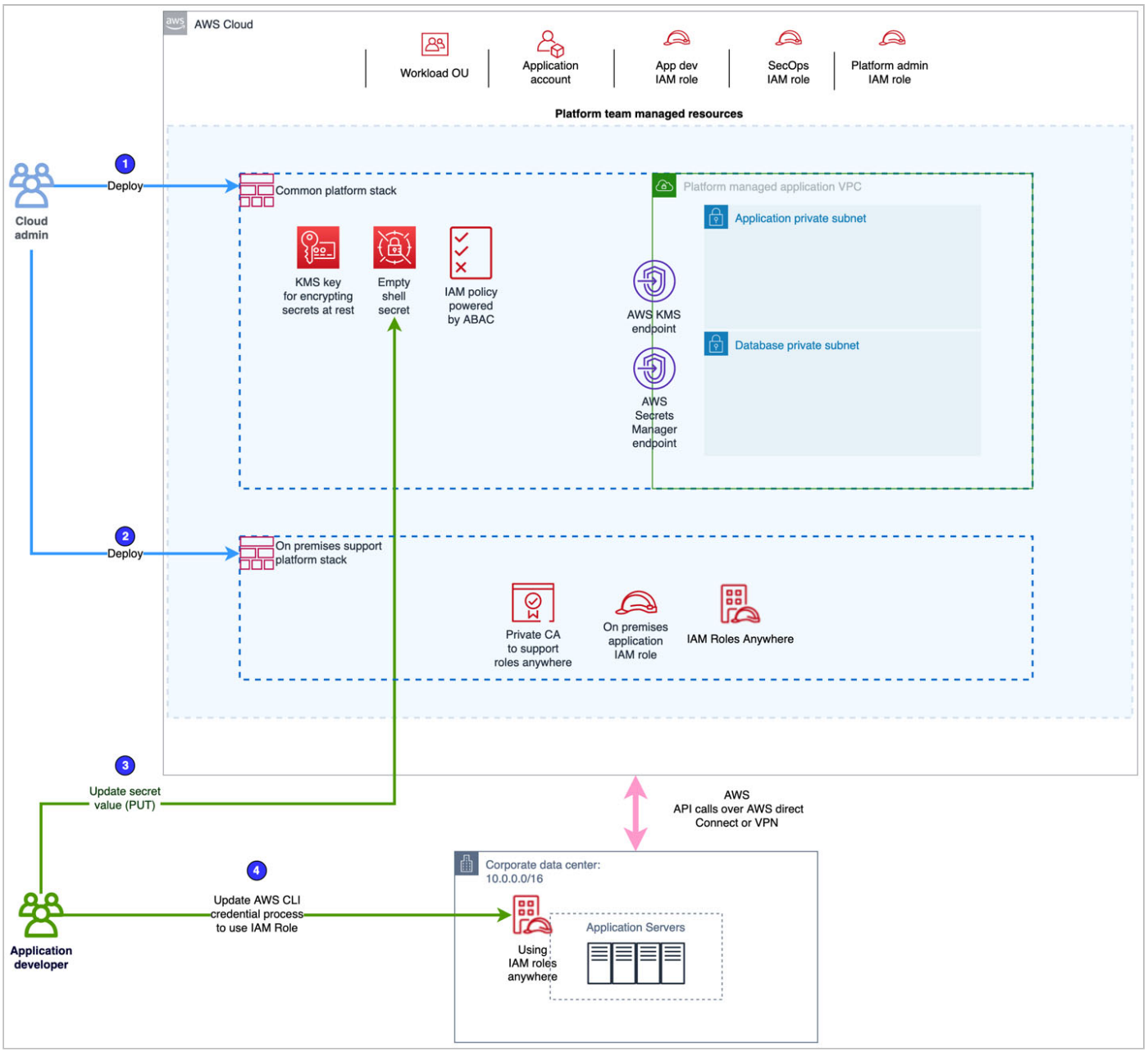

Figure 1: Application on premises connecting to Secrets Manager

To deploy Archetype 1

1. *(Actions by the application team persona)* Clone the repository and update the tagging details at `configs/tagconfig.json`.

   Note: Do not modify the tag/attributes `name/key`, only modify `value`.

2. *(Actions by the environment team persona)* Run the following command to deploy the common environment infrastructure stack.
   `./helper.sh prepare`
   Then, run the following command to deploy the IAM Roles Anywhere infrastructure stack.
   `./helper.sh on-prem`

3. *(Actions by the application team persona)* Update the secret value of the dummy secrets provided by the environment team, by using the following command.
   `./helper.sh update-secret`

Note: This command will only update the secret if it's still using the dummy value.

Then, run the following command to set up the client and server on premises.

```
./helper.sh client-profile-setup
```

Follow the command prompt. It will help you request a client certificate and update the AWS CLI profile.

Important: When you request a client certificate, make sure to supply at least one distinguished name, like `CommonName`.

The sample output should look like the following.

```
--> This role can be used by the application by using the AWS CLI profile 'developer'.
--> For instance, the following output illustrates how to access secret values by using the
--> Sample AWS CLI: aws secretsmanager get-secret-value --secret-id $SECRET_ARN --profile de
```

At this point, the client-side utility (`helper.sh client-profile-setup`) should have updated the AWS CLI configuration file with the following profile.

```
[profile developer]
region = <aws-region>
credential_process = /Users/<local-laptop-user>/.aws/aws_signing_helper credential-process
    --certificate /Users/<local-laptop-user>/.aws/client_cert.pem
    --private-key /Users/<local-laptop-user>/.aws/my_private_key.clear.key
    --trust-anchor-arn arn:aws:rolesanywhere:<aws-region>:444455556666:trust-anchor/a1b2c3
    --profile-arn arn:aws:rolesanywhere:<aws-region>:444455556666:profile/a1b2c3d4-5678-90
    --role-arn arn:aws:iam::444455556666:role/RolesanywhereabacStack-onPremAppRole-12345678
```

To test Archetype 1 deployment

- The application team can verify that the AWS CLI profile has been properly set up and is capable of retrieving secrets from Secrets Manager by running the following client-side utility command.
  `./helper.sh on-prem-test`

This client-side utility (`helper.sh`) command verifies that the AWS CLI profile (for example, developer) has been set up for IAM Roles Anywhere and can run the GetSecretValue API action to retrieve the value of the secret stored in Secrets Manager.

The sample output should look like the following.

```
--> Checking credentials ...
{
    "UserId": "AKIAIOSFODNN7EXAMPLE:EXAMPLE11111EXAMPLEEXAMPLE111111",
    "Account": "444455556666",
    "Arn": "arn:aws:sts::444455556666:assumed-role/RolesanywhereabacStack-onPremAppRole-123
}
--> Assume role worked for:
arn:aws:sts::444455556666:assumed-role/RolesanywhereabacStack-onPremAppRole-1234567890ABC
--> This role can be used by the application by using the AWS CLI profile 'developer'.
--> For instance, the following output illustrates how to access secret values by using the
--> Sample AWS CLI: aws secretsmanager get-secret-value --secret-id $SECRET_ARN --profile $
-------Output-------
{
  "password": "randomuniquepassword",
  "servertype": "testserver1",
  "username": "testuser1"
}
-------Output-------
```

Archetype 2: Application has migrated to AWS

Archetype 2 has the following requirement:

- Deploy a sample application to demonstrate how ABAC authorization works for Secrets Manager APIs.

The application, environment, and security teams work together to define a tagging strategy that will be used to restrict access to secrets. After this, the proposed workflow for each persona is as follows:

1. The *environment engineer* deploys a common environment infrastructure stack to bootstrap the AWS account with secrets and an IAM policy by using the supplied tagging requirement.
2. The *application developer* updates the secrets required by the application by using the client-side utility (`helper.sh`).
3. The *application developer* tests the sample application to confirm operability of ABAC.

Figure 2 shows the workflow for Archetype 2.

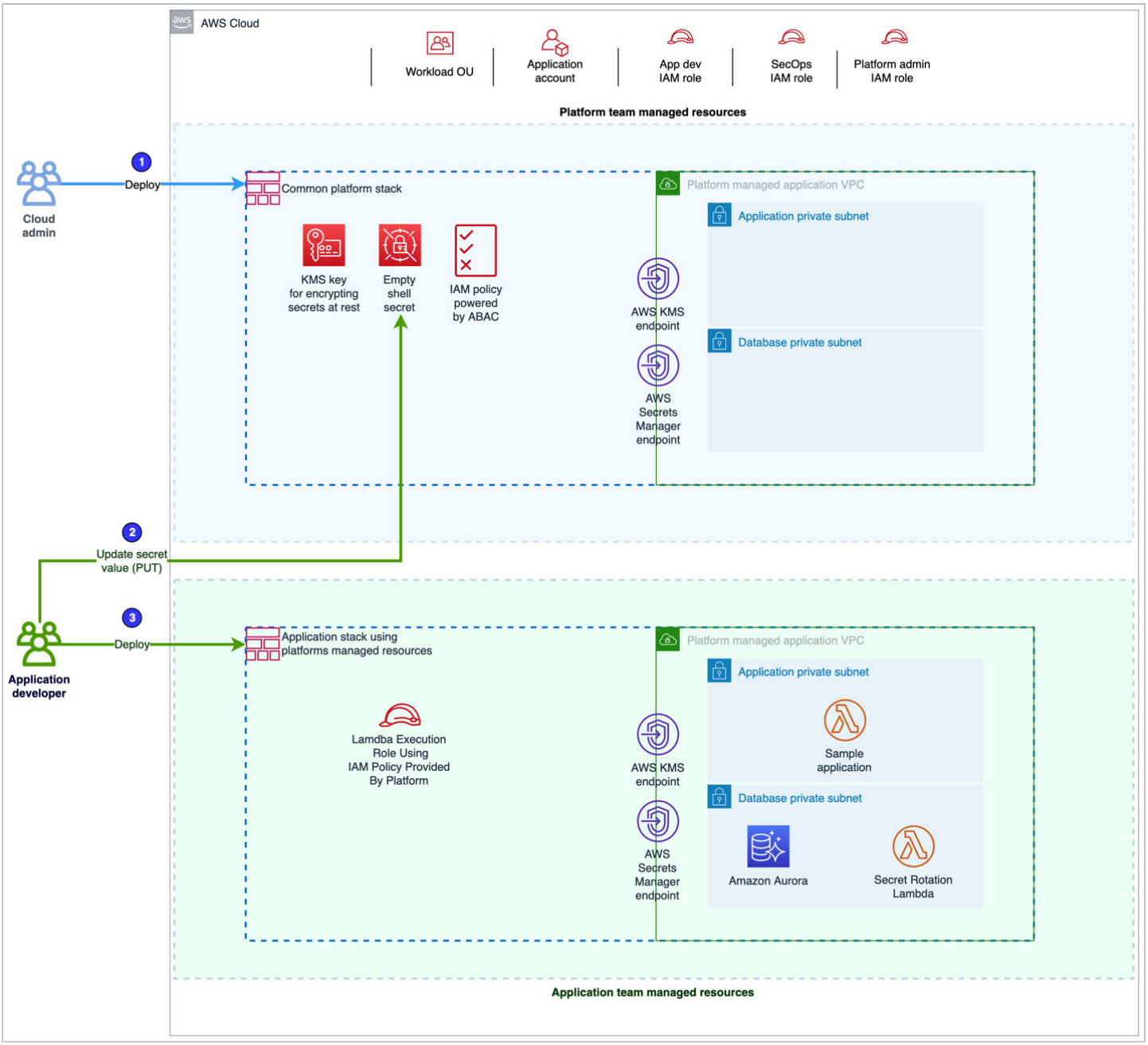

Figure 2: Sample migrated application connecting to Secrets Manager

To deploy Archetype 2

1. *(Actions by the application team persona)* Clone the repository and update the tagging details at `configs/tagconfig.json`.

   Note: Don't modify the tag/attributes name/key, only modify value.

2. *(Actions by the environment team persona)* Run the following command to deploy the common platform infrastructure stack.
   `./helper.sh prepare`

3. *(Actions by the application team persona)* Update the secret value of the dummy secrets provided by the environment team, using the following command.
   `./helper.sh update-secret`

   Note: This command will only update the secret if it is still using the dummy value.

Then, run the following command to deploy a sample app stack.

```
./helper.sh on-aws
```

> **Note:** If your secrets were migrated from a system that did not have the correct access controls, as a best security practice, you should rotate them at least once manually.

At this point, the client-side utility should have deployed a sample application Lambda function. This function connects to a MySQL database by using credentials stored in Secrets Manager. It retrieves the secret values, validates them, and establishes a connection to the database. The function returns a message that indicates whether the connection to the database is working or not.

To test Archetype 2 deployment

- The application team can use the following client-side utility (`helper.sh`) to invoke the Lambda function and verify whether the connection is functional or not.

```
./helper.sh on-aws-test
```

The sample output should look like the following.

```
--> Check if AWS CLI is installed
--> AWS CLI found
--> Using tags to create Lambda function name and invoking a test
--> Checking the Lambda invoke response.....
--> The status code is 200
--> Reading response from test function:
"Connection to the DB is working."
--> Response shows database connection is working from Lambda function using secret.
```

Conclusion

Building an effective secrets management solution requires careful planning and implementation. AWS Secrets Manager can help you effectively manage the lifecycle of your secrets at scale. We encourage you to take an iterative approach to building your secrets management solution, starting by focusing on core functional requirements like managing access, defining audit requirements, and building preventative and detective controls for secrets management. In future iterations, you can improve your solution by implementing more advanced functionalities like automatic rotation or resource policies for secrets.

To read Part 1 of this series, go to Migrating your secrets to AWS, Part I: Discovery and design.

If you have feedback about this post, submit comments in the Comments section below. If you have questions about this post, start a new thread on the AWS Secrets Manager re:Post or contact AWS Support.

Want more AWS Security news? Follow us on Twitter.

TAGS: AWS Identity and Access Management (IAM), AWS Secrets Manager, IAM Roles Anywhere, Identity, migration, secrets, Secrets Manager, Security, Security Blog

Comments