Secret Management and Credential Protection on AWS

1. Problem Statement.

A payment processing company handling sensitive financial transactions identified hardcoded credentials and inconsistent secret storage across environments. This created elevated risk exposure and compliance concerns. A centralized secrets management strategy was required to secure credentials and enforce encryption standards.

I implemented this framework to eliminate insecure storage of credentials, reduce the risk of credential compromise, and support regulatory compliance through centralized governance and automation.

2. Objectives

The primary objectives were to:

- Centralize secret storage
- Encrypt sensitive data using KMS
- Enforce least-privilege access
- Automate credential rotation
- Detect hardcoded secrets
- Prevent future credential exposure
- Improve governance and audit readiness

3. Secure Secret Storage Architecture

3.1 AWS Secrets Manager

High-risk credentials were stored in AWS Secrets Manager using encrypted key/value pairs.

Examples include:

- Database credentials
- API keys
- Service tokens

Secrets were protected using KMS encryption and standardized naming conventions.

Secrets Manager was selected for its built-in rotation, auditing, and fine-grained access control.

3.2 AWS Systems Manager Parameter Store

Application configuration secrets were stored in Parameter Store using SecureString parameters.

All parameters were encrypted using KMS and protected by IAM policies.

Naming standards were enforced to maintain consistency.

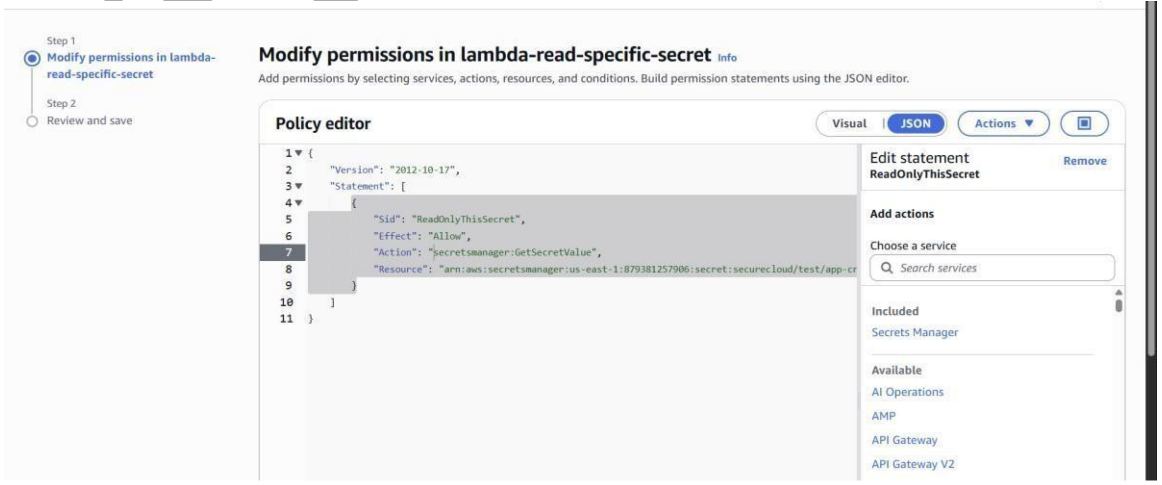
4. IAM Access Control and Least Privilege

Dedicated IAM roles were created for applications and services accessing secrets.

Policies were scoped to specific secret ARNs and parameters.

Granted permissions included:

- secretsmanager:GetSecretValue
- ssm:GetParameter



Modify permissions in lambda-read-specific-secret

Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyThisSecret",
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "arn:aws:secretsmanager:us-east-1:879381257906:secret:securecloud/test/app-cr
        }
    ]
}
```

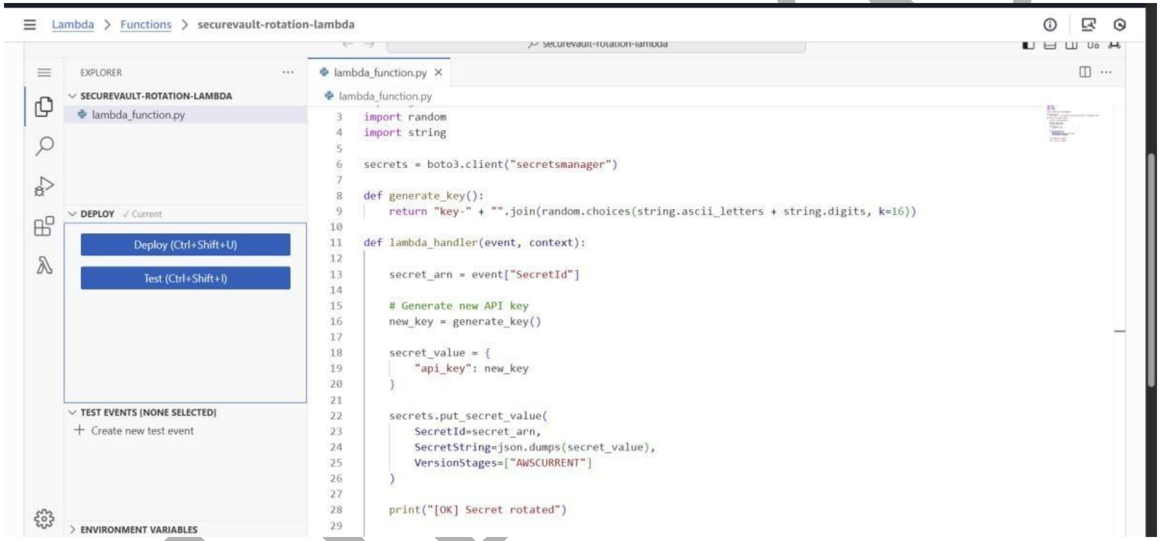
Access testing confirmed that unauthorized access attempts resulted in AccessDenied errors.

This validated least-privilege enforcement.

5. Automated Secret Rotation

5.1 Rotation Architecture

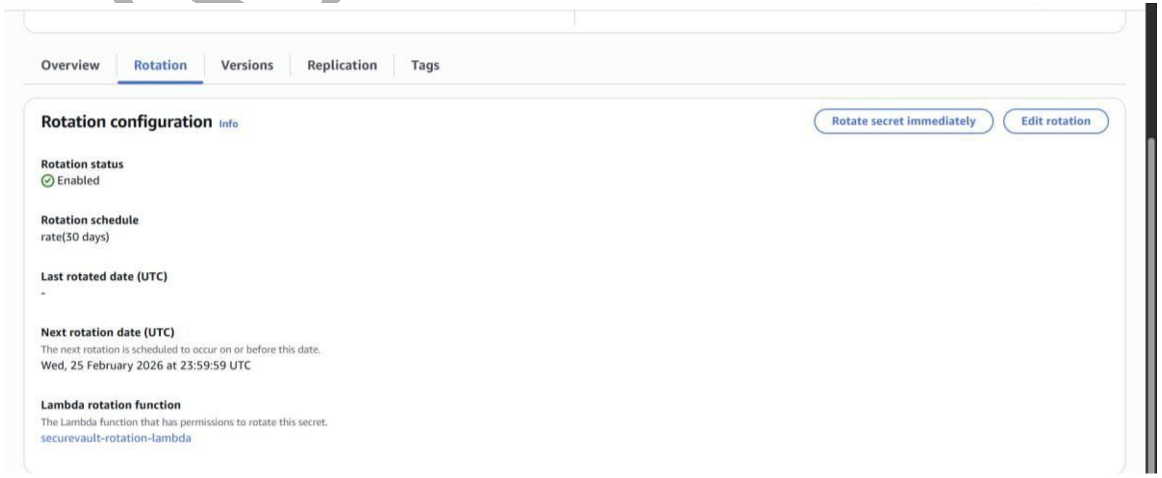
Secrets Manager was integrated with a custom Lambda rotation function.



```python
import random
import string

secrets = boto3.client("secretsmanager")

def generate_key():
    return "key-" + "".join(random.choices(string.ascii_letters + string.digits, k=16))

def lambda_handler(event, context):

    secret_arn = event["SecretId"]

    # Generate new API key
    new_key = generate_key()

    secret_value = {
        "api_key": new_key
    }

    secrets.put_secret_value(
        SecretId=secret_arn,
        SecretString=json.dumps(secret_value),
        VersionStages=["AWSCURRENT"]
    )

    print("[OK] Secret rotated")

```

The function generates new credentials and updates dependent services.

Rotation schedules were defined based on security requirements.



Rotation configuration

Rotation status
✓ Enabled

Rotation schedule
rate(30 days)

Last rotated date (UTC)
-

Next rotation date (UTC)
The next rotation is scheduled to occur on or before this date.
Wed, 25 February 2026 at 23:59:59 UTC

Lambda rotation function
The Lambda function that has permissions to rotate this secret.
securevault-rotation-lambda

5.2 Permission Configuration

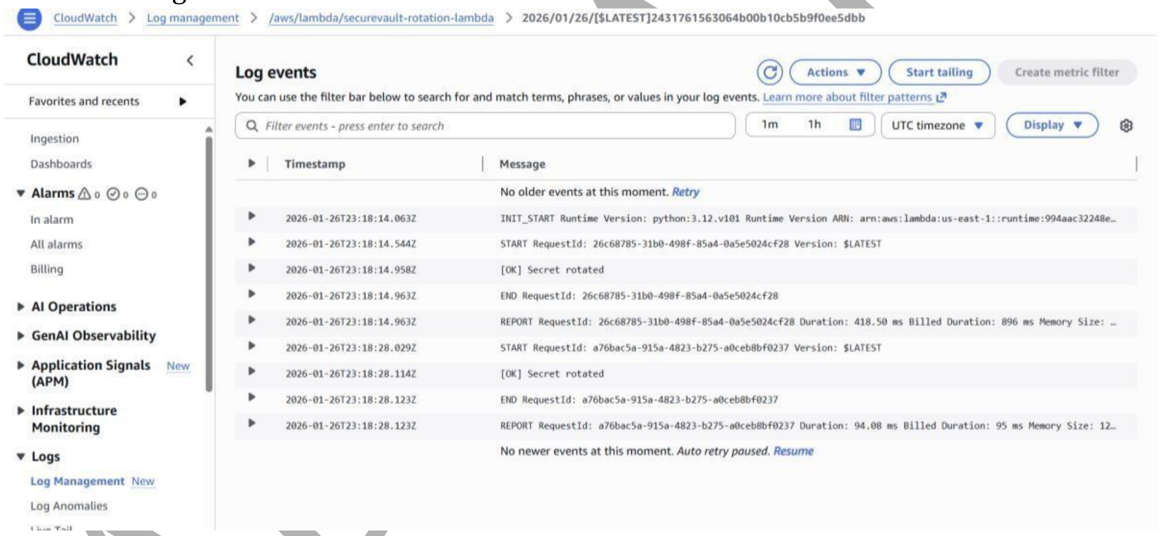
Resource-based policies were configured to allow Secrets Manager to invoke the rotation function.

Invocation was restricted to approved secret ARNs.

5.3 Rotation Validation

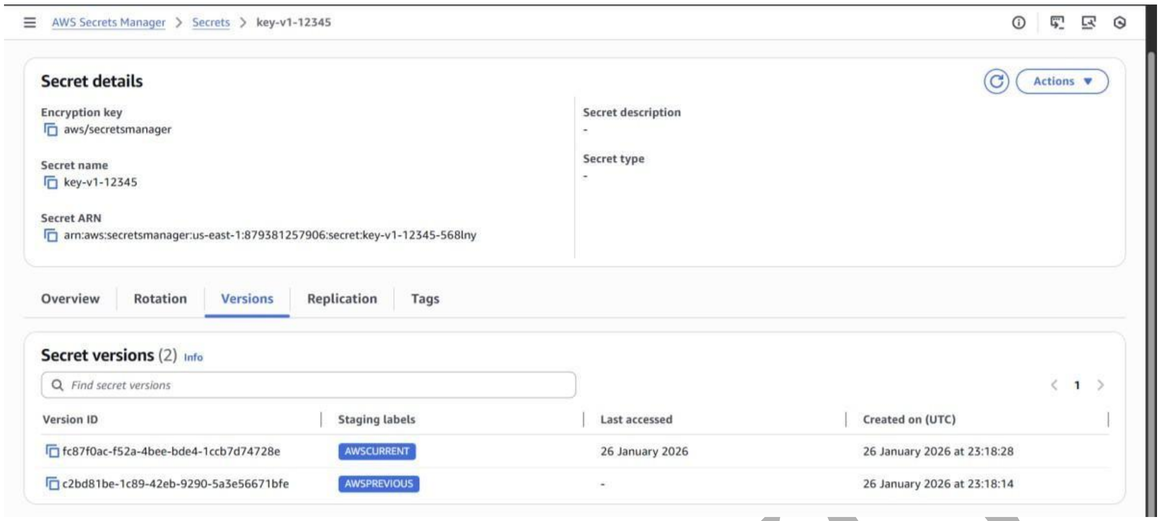
Manual and scheduled rotations were tested.

CloudWatch Logs confirmed successful execution.


CloudWatch > Log management > /aws/lambda/securevault-rotation-lambda > 2026/01/26/[$LATEST]2431761563064b00b10cb5b9f0ee5dbb
CloudWatch
Favorites and recents
Ingestion
Dashboards
Alarms 0 0 0
In alarm
All alarms
Billing
AI Operations
GenAI Observability
Application Signals (APM) New
Infrastructure Monitoring
Logs
Log Management New
Log Anomalies
Log events
Actions
Start tailing
Create metric filter
You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns
Filter events - press enter to search
1m
1h
UTC timezone
Display
Timestamp
Message

```
No older events at this moment. Retry
2026-01-26T23:18:14.063Z   INIT_START Runtime Version: python:3.12.v101 Runtime Version ARN: arn:aws:lambda:us-east-1::runtime:994aac32248e…
2026-01-26T23:18:14.544Z   START RequestId: 26c68785-31b0-498f-85a4-0a5e5024cf28 Version: $LATEST
2026-01-26T23:18:14.958Z   [OK] Secret rotated
2026-01-26T23:18:14.963Z   END RequestId: 26c68785-31b0-498f-85a4-0a5e5024cf28
2026-01-26T23:18:14.963Z   REPORT RequestId: 26c68785-31b0-498f-85a4-0a5e5024cf28 Duration: 418.50 ms Billed Duration: 896 ms Memory Size: …
2026-01-26T23:18:28.029Z   START RequestId: a76bac5a-915a-4823-b275-a0ceb8bf0237 Version: $LATEST
2026-01-26T23:18:28.114Z   [OK] Secret rotated
2026-01-26T23:18:28.123Z   END RequestId: a76bac5a-915a-4823-b275-a0ceb8bf0237
2026-01-26T23:18:28.123Z   REPORT RequestId: a76bac5a-915a-4823-b275-a0ceb8bf0237 Duration: 94.08 ms Billed Duration: 95 ms Memory Size: 12…
No newer events at this moment. Auto retry paused. Resume
```

New version stages were created automatically after rotation.



Secret details

Encryption key
aws/secretsmanager

Secret name
key-v1-12345

Secret ARN
arn:aws:secretsmanager:us-east-1:879381257906:secret:key-v1-12345-568lny

Secret description
-

Secret type
-

Secret versions (2) Info

Version ID	Staging labels	Last accessed	Created on (UTC)
fc87f0ac-f52a-4bee-bde4-1ccb7d74728e	AWSCURRENT	26 January 2026	26 January 2026 at 23:18:28
c2bd81be-1c89-42eb-9290-5a3e56671bfe	AWSPREVIOUS	-	26 January 2026 at 23:18:14

This ensures continuous credential freshness.

6. Hardcoded Secret Detection

6.1 Scanning Tool Implementation

TruffleHog was implemented as the primary secret scanning tool.

Docker-based execution was used for consistency across environments.

Scans analyzed repositories for:

- Credential patterns
- High-entropy strings
- Provider tokens

6.2 Detection and Analysis

Test credentials were detected successfully during scanning.

Findings were documented and classified by severity.

6.3 Remediation

All exposed secrets were removed from source code.

Secure references to Secrets Manager and Parameter Store replaced hardcoded values.

Follow-up scans confirmed remediation.

7. Preventive Controls

7.1 Pre-Commit Enforcement

Git pre-commit hooks were implemented to run TruffleHog before commits.

Commits containing sensitive data were blocked automatically.

7.2 Validation Testing

Controlled tests confirmed that insecure commits were prevented.

Only sanitized code was permitted.

8. Governance and Documentation

All secret management procedures, access policies, and rotation workflows were documented.

Documentation supports:

- Security audits
- Compliance reviews
- Incident investigations
- Operational continuity

9. Testing and Validation Framework

Security controls were validated through:

- Secret retrieval testing
- Permission reduction testing
- Rotation testing
- Detection scanning
- Pre-commit enforcement testing

All controls functioned as designed.

10. Outcomes and Impact

This implementation delivered:

- Centralized credential governance
- Reduced credential exposure risk
- Automated rotation capability
- Improved code hygiene
- Strong audit readiness
- Enhanced security posture

11. Conclusion

I designed and implemented an enterprise-grade secret management framework on AWS.

Through secure storage, automated rotation, preventive scanning, and strict access control, this solution ensures sensitive credentials are protected throughout their lifecycle.