IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15313

Secure Docker Containers with AWS IAM and AWS Secrets Manager

C. B. Greeshma[1], Mr. B. Ramesh Kumar[2]

III BCA, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore, Tamil Nadu, India.[1]

Assistant Professor, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore, Tamil Nadu, India.[2]

Abstract: Containerization has become a fundamental approach in modern cloud computing, offering portability, scalability, and operational efficiency for application deployment. However, securing containerized applications remains a critical challenge, particularly when managing sensitive data such as API keys, database credentials, and authentication tokens. This project focuses on implementing robust security practices for Docker containers deployed in cloud environments by integrating AWS Identity and Access Management (IAM) and AWS Secrets Manager. IAM is used to enforce fine-grained access control through roles and policies, ensuring that containers operate under the principle of least privilege. AWS Secrets Manager provides a secure mechanism for storing, managing, and dynamically retrieving sensitive credentials without hardcoding them into application code or Docker images. The implementation demonstrates a secure and scalable architecture that eliminates credential exposure risks, enhances compliance with cloud security standards, and improves overall system reliability. Results confirm that the system successfully achieves zero hardcoded credentials, encrypted secret storage, role-based access control, and automatic secret rotation capability, providing a reproducible reference architecture for securing containerized cloud-native applications.

Keywords: Docker, AWS IAM, AWS Secrets Manager, Container Security, DevOps, Cloud Security, Least Privilege, Credential Management.

I. INTRODUCTION

The rapid evolution of cloud computing and microservices architecture has made containerization a cornerstone technology for modern application deployment. Docker enables developers to package applications along with their dependencies into lightweight, portable containers that execute consistently across diverse environments, from local development workstations to production cloud infrastructure. This approach significantly improves scalability, simplifies deployment workflows, and enhances resource utilization across organizations of all sizes.

However, as container adoption continues to accelerate, security concerns related to identity management, access control, and sensitive credential handling have become increasingly critical. One of the most prevalent security risks in containerized environments is the improper management of sensitive information including database passwords, API keys, and authentication tokens. Hardcoding such credentials within Docker images or configuration files creates serious vulnerabilities that can be exploited when images are shared or containers are compromised.

Traditional approaches often grant excessive permissions to containers, violating the security principle of least privilege and significantly expanding the potential attack surface. The absence of centralized credential management further compounds these risks, making secret rotation and revocation complex and error-prone operations.

To address these challenges, Amazon Web Services provides cloud-native security services that can be seamlessly integrated into containerized application workflows. AWS Identity and Access Management (IAM) allows precise definition of access policies and dynamic role assignment, ensuring that containers operate with only the minimum permissions required. AWS Secrets Manager provides a secure, centralized mechanism for storing, managing, and automatically rotating sensitive credentials without embedding them into application code.

This project designs and implements a secure Docker-based application environment by integrating AWS IAM and AWS Secrets Manager, demonstrating how container security can be substantially enhanced through proper identity management and secure secret handling, thereby reducing vulnerabilities and improving system reliability in cloud deployments.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15313

II. RELATED WORK

Research on containerization security has grown substantially alongside the widespread adoption of Docker in enterprise and cloud-native environments. Traditional application deployment approaches involve storing credentials within configuration files or environment variables, practices that introduce significant security risks when applied to containerized workloads. Studies consistently identify credential exposure through source code repositories and shared Docker images as among the most frequent causes of cloud security incidents.

The DORA (DevOps Research and Assessment) metrics framework demonstrates that organizations implementing automated security controls and least-privilege access achieve lower change failure rates and faster recovery from security incidents. Research on cloud-native identity management confirms that role-based access control significantly reduces the attack surface compared to static credential approaches.

AWS IAM has been widely studied as a comprehensive identity management platform that provides fine-grained permission control through JSON-based policy definitions. Unlike manual credential management approaches, IAM enables temporary security credentials that are automatically rotated, eliminating the long-term exposure risks associated with static access keys.

AWS Secrets Manager has been examined in the literature as a purpose-built service for secure credential storage with built-in encryption, access auditing, and automatic rotation capabilities. Existing literature identifies a gap in practical, end-to-end documented implementations combining Docker containerization with AWS IAM role assignment and Secrets Manager integration. This project addresses that gap by providing a complete, reproducible implementation with comprehensive documentation and validated results.

III. OBJECTIVES AND CHALLENGES

The primary objectives of this project are: (1) to implement a secure containerized application environment using Docker with no hardcoded credentials; (2) to integrate AWS Identity and Access Management for role-based access control enforcing the principle of least privilege; (3) to configure AWS Secrets Manager for centralized, encrypted credential storage and dynamic runtime retrieval; (4) to demonstrate automated secret rotation and access logging capabilities within the cloud security architecture; and (5) to provide a reproducible reference implementation with comprehensive documentation for cloud security practitioners.

Development Challenges

The primary technical challenge involved configuring IAM role assignment for Docker containers running outside managed AWS container orchestration services. Granting containers the ability to assume IAM roles required careful configuration of instance profiles and trust relationships, ensuring that only authorized containers could request temporary credentials through the AWS Security Token Service.

Integrating the AWS SDK within the containerized Python Flask application required managing region configuration, secret identifier references, and exception handling for scenarios including expired credentials, unauthorized access attempts, and network failures. The application was designed to retrieve secrets dynamically at startup without caching credentials permanently, ensuring that secret rotations are reflected without requiring container restarts.

Ensuring that no sensitive information was embedded in Docker images required disciplined implementation of the .dockerignore file and strict code review processes. All secret identifiers and region configurations were passed through non-sensitive environment variables, while actual credential values were retrieved exclusively through authenticated Secrets Manager API calls at runtime.

IV. SYSTEM ARCHITECTURE

The system follows a secure cloud-integrated container architecture where the application layer, identity management, and secret storage operate as distinct, loosely coupled components. The Docker container hosts the application logic and communicates exclusively with AWS services through authenticated HTTPS API calls. IAM roles define the permission boundaries of each container, while Secrets Manager provides the centralized credential repository.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15313

Table 1. System Architecture Layers

Layer	Technology	Role
Application Layer	Python Flask, Docker, AWS SDK (Boto3)	Application logic and secure secret retrieval
Containerization	Docker, Dockerfile	Packages application and runtime into portable, isolated images
Identity Management	AWS IAM Roles & Policies	Role-based access control enforcing least privilege
Secret Storage	AWS Secrets Manager	Encrypted storage and dynamic retrieval of credentials
Compute Infrastructure	AWS EC2 (Ubuntu 22.04)	Hosts Docker Engine and running application containers
Security Communication	HTTPS / TLS Encryption	Encrypted channel between container and AWS services
Monitoring & Audit	AWS CloudTrail, Container Logs	Tracks secret access, IAM role usage, and security events
End Users	HTTP via EC2 Public IP	Access to deployed Flask application

The architecture adheres to the principle of infrastructure immutability and credential externalization. Container images are built without any embedded credentials, and all sensitive values are retrieved exclusively at runtime through authenticated API calls to Secrets Manager. This design ensures that even if a container image is compromised or shared, no credential exposure occurs.

IAM policies are defined using the JSON policy language, granting containers permission only to retrieve specific named secrets from Secrets Manager. The trust relationship defined in the IAM role ensures that only containers running on authorized EC2 instances can assume the role and request temporary security credentials.

V. IMPLEMENTATION

The implementation is organized into five functional modules, each addressing a distinct layer of the secure container architecture. This modular design ensures that individual components can be independently updated, tested, and replaced without disrupting the overall system security posture.

A. Python Flask Web Application

The application is built using Python 3.11 and the Flask micro-framework, implementing a REST API that demonstrates secure secret retrieval from AWS Secrets Manager. The application uses the Boto3 AWS SDK to authenticate via the assigned IAM role and request the database credentials secret at runtime. A /health endpoint is included for deployment validation, and structured error handling ensures that exceptions are logged without exposing sensitive credential values in responses or logs.

B. Docker Containerization

The application is packaged into a Docker container using a Dockerfile based on the official Python 3.11 slim image. Security best practices are enforced throughout the image build process: a non-root user is created for application execution, only required files are copied into the image, and a. dockerignore file excludes local configuration files, environment files, and Git metadata from the build context. No credentials, secret names, or sensitive configuration values are embedded within the Dockerfile or the resulting image.

C. AWS IAM Configuration

An IAM role is created specifically for the Docker container with a trust policy allowing EC2 instances to assume the role. A custom IAM policy is attached that grants permission exclusively to the secretsmanager: GetSecretValue action on the specific secret ARN, following the principle of least privilege. No wildcard resource access or overly broad permissions are granted. The IAM role is attached to the EC2 instance hosting the Docker container, enabling the application to obtain temporary security credentials automatically through the Instance Metadata Service.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15313

D. AWS Secrets Manager Configuration

The database credentials secret is created in AWS Secrets Manager in JSON format, storing the username, password, host, and port required for database connectivity. AWS-managed encryption keys are used to encrypt the secret at rest. The application references the secret by its name identifier, and IAM policy verification occurs automatically before each retrieval request. Optional automatic rotation is configured to periodically update credentials without requiring application restarts or manual intervention.

E. Application Security and Error Handling

The application implements structured exception handling for all Secrets Manager API calls, distinguishing between AccessDeniedException for IAM authorization failures, ResourceNotFoundException for invalid secret identifiers, and general network or service errors. Error responses are formatted to provide sufficient debugging context without revealing sensitive credential values or internal system architecture details. All communications between the container and AWS services occur over HTTPS with TLS encryption.

VI. RESULTS AND EVALUATION

The secure container system was tested comprehensively across authentication, secret retrieval, access control enforcement, and application performance dimensions. All test scenarios executed successfully, confirming the reliability and security of the implemented architecture.

Table 2. System Performance and Security Results

Metric	Measured Result	Benchmark	Status
Secret Retrieval Latency	< 120 ms	< 500 ms	✓ PASS
Container Startup Time	< 8 sec	< 30 sec	✓ PASS
Application Response Time	< 60 ms	< 200 ms	✓ PASS
Docker Image Size	142 MB	< 500 MB	✓ PASS
Hardcoded Credentials	Zero	Zero	✓ PASS
Unauthorized Access Blocked	100%	100%	✓ PASS
Secret Encryption at Rest	Enabled	Required	✓ PASS
Automatic Secret Rotation	Configured	Supported	✓ PASS

Table 3. Security Test Scenarios

Test Scenario	Expected Result	Outcome
Container startup with valid IAM role	Secret retrieved successfully	PASS
Container with no IAM role	AccessDeniedException returned	PASS
Invalid secret name provided	ResourceNotFoundException returned	PASS
Docker image inspection for credentials	No credentials found in image	PASS
Application HTTP endpoint accessible	Flask app returns 200 OK	PASS
Secret rotation applied	Updated credentials retrieved	PASS
Excessive permission policy rejected	Least privilege policy enforced	PASS
Container logs inspected for secrets	No sensitive values in logs	PASS

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026
DOI: 10.17148/IJARCCE.2026.15313

Authentication testing confirmed that containers operating under the assigned IAM role successfully retrieved secrets while containers without the role received structured AccessDeniedException responses. The IAM policy boundary enforcement was verified across multiple access scenarios, confirming that no unauthorized resource access occurred in any test case.

Performance evaluation demonstrated that secret retrieval latency remains well within acceptable bounds, adding less than 120 milliseconds to application startup time. The containerized application was accessible via the EC2 public IP immediately following container startup, with the /health endpoint returning HTTP 200 responses confirming successful initialization and secret retrieval.

The project demonstrates measurable security improvements across all key metrics. Credential exposure risk was eliminated through zero hardcoded secrets. Access control was enforced through least-privilege IAM policies. Secret lifecycle management was automated through Secrets Manager rotation, and audit visibility was provided through CloudTrail logging of all secret access events.

VII. CONCLUSION

This project successfully implemented a secure containerized application environment using Docker integrated with AWS Identity and Access Management and AWS Secrets Manager for a Python Flask web application hosted on AWS EC2. The architecture effectively eliminates the most prevalent container security vulnerabilities by removing hardcoded credentials, enforcing role-based access control, and centralizing secret management through cloud-native AWS security services.

Performance evaluation confirms that security controls introduce negligible latency overhead, with secret retrieval completing in under 120 milliseconds and total application startup in under eight seconds. All eight security test scenarios passed, demonstrating the reliability of the IAM authentication, Secrets Manager integration, and access denial mechanisms. The implementation adheres to cloud security best practices including the principle of least privilege, encrypted credential storage, secure HTTPS communication, and comprehensive audit logging.

The results validate that combining Docker containerization with AWS IAM and Secrets Manager provides a scalable, reliable, and cost-efficient foundation for securing cloud-native applications. The documented architecture, configuration files, IAM policy structures, and application code serve as a reproducible reference that security practitioners and DevOps engineers can adapt for production-grade implementations.

VIII. FUTURE ENHANCEMENTS

Several enhancements can extend the capabilities and maturity of this implementation. The most significant improvement would be migration to container orchestration using Amazon ECS or Amazon EKS, enabling automatic scaling, self-healing deployments, and native IAM task role assignment that simplifies credential management at scale without requiring EC2 instance profile configuration.

Integration with a CI/CD pipeline using GitLab CI/CD or AWS CodePipeline would automate Docker image security scanning, IAM policy validation, and container deployment, ensuring that security configurations are continuously validated with every code change. Automated container image vulnerability scanning using tools such as Amazon ECR image scanning or Trivy would identify security vulnerabilities in base images before deployment.

Implementation of a Zero Trust security model would further strengthen the architecture by continuously verifying every inter-service request rather than relying solely on network boundaries. Advanced monitoring using Amazon CloudWatch and AWS Security Hub would provide real-time visibility into secret access patterns, IAM role usage anomalies, and container runtime behavior, enabling proactive threat detection and incident response.

Infrastructure as Code using AWS CloudFormation or Terraform would automate the provisioning of all IAM roles, policies, Secrets Manager secrets, and EC2 infrastructure, ensuring reproducible and version-controlled security configurations across development, staging, and production environments.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026
DOI: 10.17148/IJARCCE.2026.15313

REFERENCES

[1]. Amazon Web Services. (2024). AWS Identity and Access Management User Guide. https://docs.aws.amazon.com/IAM/latest/UserGuide/
[2]. Amazon Web Services. (2024). AWS Secrets Manager Developer Guide. https://docs.aws.amazon.com/secretsmanager/latest/userguide/
[3]. Docker Documentation. (2024). Docker Engine Overview. https://docs.docker.com/get-started/
[4]. Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, 2014(239).
[5]. Wittig, A., & Wittig, M. (2023). Amazon Web Services in Action (3rd ed.). Manning Publications.
[6]. Amazon Web Services. (2024). Amazon EC2 User Guide for Linux Instances. https://docs.aws.amazon.com/ec2/
[7]. Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps. IT Revolution Press.
[8]. Cloud Security Alliance. (2023). Best Practices for Managing Secrets in Cloud-Native Applications. CSA Research Publication.
[9]. Kim, G., Humble, J., Debois, P., & Willis, J. (2016). The DevOps Handbook. IT Revolution Press.
[10]. Sharma, D., et al. (2020). Continuous Integration and Deployment: Automation for Faster Software Delivery. International Journal of Computer Applications, 175(12), 1–6.

