https://doi.org/10.31891/2219-9365-2026-87-32
UDC 004.056:004.75

MUKHA Dmytro
DarkCloud, Lead Product Security & DevSecOps Engineer
https://orcid.org/0009-0002-2155-2739
e-mail: muha.dimaa@gmail.com

LAMBDA ENVIRONMENT VARIABLE SECURITY: INCIDENT INVESTIGATION, MASS CREDENTIAL ANALYSIS, AND ROTATION METHODOLOGY IN AWS CLOUD ENVIRONMENTS

AWS Lambda is the dominant serverless compute platform, executing over one trillion function invocations per month across enterprise cloud environments. This paper investigates a critical, underexplored attack surface: plaintext credential exposure through the Lambda GetFunction API. When an attacker obtains high-privilege AWS credentials, every environment variable stored inline across all deployed Lambda functions becomes immediately accessible in a single automated sweep. This paper presents a forensic reconstruction of a real-world incident in which a compromised root access key enabled the exfiltration of credentials from ~1000 Lambda functions within 50 minutes. We describe a structured investigation methodology, present open-source tooling for mass credential classification, and propose a prioritised rotation and migration framework based on AWS Security Best Practices. Our findings demonstrate that environment variable storage in Lambda constitutes a systemic architectural vulnerability that requires organisational remediation beyond individual credential rotation.

Keywords: AWS Lambda, cloud security, credential exposure, secrets management, incident response

МУХА Дмитро
DarkCloud, провідний інженер з безпеки продуктів та DevSecOps

БЕЗПЕКА ЗМІННИХ СЕРЕДОВИЩА AWS LAMBDA: РОЗСЛІДУВАННЯ ІНЦИДЕНТУ, МАСОВИЙ АНАЛІЗ ОБЛІКОВИХ ДАНИХ ТА МЕТОДОЛОГІЯ ЇХ РОТАЦІЇ У ХМАРНИХ СЕРЕДОВИЩАХ AWS

AWS Lambda є домінувальною платформою безсерверних обчислень, яка щомісяця виконує понад трильйон викликів функцій у корпоративних хмарних середовищах. У статті досліджено критичну та недостатньо вивчену поверхню атаки - розкриття облікових даних у відкритому вигляді через API GetFunction сервісу Lambda. За наявності у зловмисника високопривілейованих облікових даних AWS кожна змінна середовища, збережена безпосередньо в конфігурації розгорнутих функцій Lambda, стає негайно доступною під час єдиного автоматизованого сканування. Наведено криміналістичну реконструкцію реального інциденту, у якому скомпрометований кореневий ключ доступу уможливив ексфільтрацію облікових даних із приблизно 1000 функцій Lambda протягом 50 хвилин. Атака охопила паролі баз даних, ключі платіжних систем, секрети підписання JWT та облікові дані сторонніх сервісів, що загалом становило повну компрометацію облікової поверхні застосунку. У роботі представлено структуровану методологію розслідування на основі журналів AWS CloudTrail, відкритий інструментарій для масової класифікації облікових даних за шаблонами імен без розкриття їхніх значень, а також матрицю пріоритезації ротації (P0–P3). Обґрунтовано пріоритетний механізм ротації та міграції секретів відповідно до найкращих практик безпеки AWS, зокрема перенесення секретів зі змінних середовища до AWS Secrets Manager із їх отриманням під час «холодного» старту та кешуванням у середовищі виконання. Розглянуто також посилення політик IAM та превентивні засоби виявлення (GuardDuty, AWS Config, CloudWatch, сканування секретів). Отримані результати демонструють, що зберігання секретів у змінних середовища Lambda є системною архітектурною вразливістю, яка потребує організаційного усунення поза межами ротації окремих облікових даних та координації на рівні організації.

Ключові слова: AWS Lambda, хмарна безпека, розкриття облікових даних, керування секретами, реагування на інциденти

Стаття надійшла до редакції / Received 17.07.2026
Прийнята до друку / Accepted 17.08.2026
Опубліковано / Published 10.09.2026

INTRODUCTION

Cloud-native architectures increasingly delegate application logic to serverless functions. AWS Lambda, Amazon Web Services' Function-as-a-Service platform, executes stateless code in response to events without requiring the provisioning or management of underlying server infrastructure. Lambda functions are invoked by API Gateway, S3 events, CloudWatch schedules, SQS queues, and dozens of other AWS services, making them the connective tissue of modern cloud-native applications (Amazon Web Services, 2024).

A common and convenient pattern for configuring Lambda functions is the injection of configuration values through environment variables. This mechanism allows developers to parameterise function behaviour across environments (development, staging, production) without modifying source code. However, the same convenience that makes environment variables attractive for configuration also makes them a high-value target when secrets are

stored inline. Unlike application code, which is typically version-controlled and subject to code review, Lambda environment variables bypass standard software development lifecycle controls entirely (Marin, Perino and Di Pietro, 2022; Datadog Security Labs, 2024).

This paper is motivated by a real-world incident in which a compromised AWS root access key was used to systematically enumerate and exfiltrate credentials from ~1000 Lambda functions across a production cloud environment. The attack exploited the AWS CloudTrail-documented GetFunction API, which returns all environment variable values in plaintext in its response. The incident exposed database passwords, payment processing API keys, JWT signing secrets, and third-party service credentials, collectively representing a complete compromise of the application's credential surface (Amazon Web Services, 2023; MITRE Corporation, 2024).

The contributions of this paper are: (1) a description of the Lambda environment variable threat model in the context of credential exfiltration; (2) a forensic methodology for investigating such incidents using AWS CloudTrail; (3) an automated toolchain for mass credential classification across large Lambda deployments; (4) a structured rotation and migration framework; and (5) architectural recommendations for eliminating the vulnerability class entirely.

The threat described in this paper carries significant implications beyond commercial cloud environments. Cloud infrastructure increasingly underpins military logistics, governmental digital services, intelligence systems, and critical national infrastructure. A credential exfiltration attack against a serverless backend hosting defence-adjacent workloads - command-and-control APIs, sensor data ingestion pipelines, or logistics coordination systems - could expose the entire operational credential surface of a deployed system within a single automated sweep. In the context of hybrid warfare, adversarial actors routinely combine cyber operations against digital infrastructure with kinetic effects, making the rapid exfiltration and weaponisation of cloud credentials a viable strategic instrument (Hutchins, Cloppert and Amin, 2011; Rid and Buchanan, 2015). The attack pattern documented in this paper - systematic enumeration of Lambda functions using a compromised high-privilege credential - is trivially adaptable to defence and governmental cloud deployments and warrants attention from cyber defence policymakers and military cloud architects. Supranational frameworks for continuous serverless security monitoring, coordinated credential hygiene standards, and cross-sector incident reporting should explicitly address the systemic Lambda environment variable vulnerability documented here.

ACKGROUND: AWS LAMBDA AND ENVIRONMENT VARIABLE SECURITY

1 AWS Lambda Architecture

AWS Lambda is a serverless compute service that runs code in response to triggers and automatically manages the underlying compute resources. A Lambda function consists of a deployment package (source code and dependencies), a runtime (Node.js, Python, Java, Go, etc.), an execution role (an IAM role defining the function's AWS permissions), and a configuration block containing memory allocation, timeout, concurrency settings, and environment variables.

Lambda functions are executed in micro-virtual machine (MVM) environments managed by AWS using the Firecracker hypervisor (Amazon Web Services, 2024). Each execution environment is initialised on cold start and may be reused for subsequent invocations (warm start). The execution environment lifecycle is fundamental to understanding the security implications of environment variable storage: all environment variables are loaded into memory during the cold start initialisation phase and remain in the execution environment for its lifetime.

2 The Environment Variable Threat Model

AWS encrypts Lambda environment variables at rest using a customer-managed or AWS-managed KMS key. This encryption protects against physical media access and certain categories of insider threat at the infrastructure level (Amazon Web Services, 2024). However, this encryption is transparent to the Lambda runtime and to the GetFunction API: when a caller with sufficient IAM permissions invokes GetFunction, the API response contains a decrypted, plaintext representation of all environment variables (Marin, Perino and Di Pietro, 2022).

The required IAM permission for this operation is lambda:GetFunction. In many organisations, this permission is granted broadly to CI/CD service accounts, developer roles, and administrative users. In the worst case - exemplified by the incident described in this paper - root-level credentials provide unrestricted access to this API without any resource-level restriction.

The attack surface is compounded by the scale of modern Lambda deployments. An organisation with hundreds of Lambda functions, each potentially containing distinct sets of credentials for different services, presents an attacker with a comprehensive credential map of the entire application architecture in a single automated API sweep.

INCIDENT DESCRIPTION AND FORENSIC ANALYSIS

1 Incident Overview

The incident analysed in this paper occurred in March 2026 and affected a production AWS environment running a mobile application backend with ~1000 deployed Lambda functions across multiple functional domains:

user authentication, payment processing, marketing automation, conversion tracking, and administrative operations. The environment used a single AWS account in the us-east-1 region.

The attack was initiated following the compromise of the AWS root access key. The key had been identified by the TruffleHog automated secret scanner, from different IP addresses. AWS issued a risk notification; this notification was not actioned by the operations team. The primary attack began.

2 Attack Methodology and Timeline

CloudTrail analysis of the incident reveals a systematic, automated reconnaissance operation. The attacker's tooling used the aws-sdk-go-v2/1.41.3 user agent on Linux/amd64, consistent with a compiled Go-based reconnaissance framework such as CloudFox or a custom credential harvesting tool.

The attack proceeded in four phases: credential validation via GetCallerIdentity; parallel enumeration of 36 AWS services within 60 seconds; targeted secret exfiltration from Secrets Manager (17+ secrets, including production database credentials); and systematic Lambda function enumeration via ListFunctions followed by individual GetFunction calls on all ~1000 functions, exposing every environment variable across the entire deployment.

The Lambda exfiltration phase is of particular interest. The attacker called GetFunction, GetFunctionUrlConfig, and GetPolicy for each of the ~1000 functions, generating ~3000 Lambda-related CloudTrail events. The entirety of the environment variable credential surface was exfiltrated within this 1-hour window.

3 CloudTrail Investigation Methodology

CloudTrail provides a complete audit log of AWS API calls, including caller identity, source IP, event name, request parameters, and response elements (Amazon Web Services, 2023). For Lambda incident investigation, the following CloudTrail event names are of primary forensic interest (MITRE Corporation, 2024):

```
GetFunction20150331v2          # Returns full function config incl. env vars
GetFunctionUrlConfig           # Returns function URL configuration
ListFunctions20150331          # Enumerates all functions in account
GetPolicy20150331v2            # Returns resource-based policy
UpdateFunctionConfiguration    # Modification of env vars (persistence)
DeleteFunction20150331         # Destructive action
```

Investigators should extract all Lambda events from CloudTrail using the AWS CLI or CloudTrail Lake SQL queries. The following command exports all Lambda-related events from a specified time window:

```
aws cloudtrail lookup-events \
    --lookup-attributes AttributeKey=EventSource,\
    AttributeValue=lambda.amazonaws.com \
    --start-time 2026-03-13T00:00:00Z \
    --end-time 2026-03-14T00:00:00Z \
    --region us-east-1 \
    --output json > lambda cloudtrail events.json
```

Once exported, investigators should identify the volume of GetFunction calls per source IP, the time distribution of calls (automated sweeps exhibit characteristic uniform inter-call timing), and the caller identity (IAMUser, AssumedRole, or Root). A high volume of GetFunction calls from a single IP address in a short window is a reliable indicator of automated credential harvesting.

MASS CREDENTIAL ANALYSIS METHODOLOGY

1 Environment Variable Export

The first step in post-incident credential analysis is the generation of a comprehensive environment variable inventory. AWS CloudShell provides a convenient, authenticated execution environment for this purpose. The following command exports the configuration of all Lambda functions in parallel, leveraging xargs for concurrency:

```
REGION="us-east-1"
aws lambda list-functions --region $REGION --output json | \
  jq -r '.Functions[].FunctionName' | \
  xargs -P10 -I{} aws lambda get-function-configuration \
    --function-name {} \
    --query '[FunctionName, Environment.Variables]' \
    --output json >> lambda envs ${REGION}.json
```

The -P10 flag instructs xargs to execute up to 10 concurrent API calls, significantly reducing total execution time for large function inventories. For a deployment of ~1000 functions, this process completes in approximately 3-4 minutes depending on network latency. The output is a JSON array of [FunctionName, {Variables}] tuples.

This procedure should be repeated for each active AWS region. Organisations may have Lambda functions deployed in multiple regions, each maintaining independent environment variable configurations. The forensic and remediation scope must include all regions where the attacker had API access.

2 Automated Credential Classification

Manual review of environment variables across hundreds of functions is impractical. The following Python classification script applies regular expression patterns to identify credentials by key name, without exposing secret values:

```
import json, re, csv

CRITICAL = re.compile(r'(?i)(password|passwd|secret.?key|
    private.?key|jwt|stripe.?secret|db.?pass|database.?url)')
HIGH = re.compile(r'(?i)(api.?key|token|auth|firebase.?key|
    reteno|appsflyer|zendesk|amplitude|hmac|webhook.?secret)')
SAFE = re.compile(r'(?i)(^stage$|^region$|log.?level|
    node.?env|_arn$|_url$|_name$|_id$|feature_)')

def classify(key):
    if SAFE.search(key): return None
    if CRITICAL.search(key): return 'P0'
    if HIGH.search(key): return 'P1'
    return 'REVIEW'
```

The classification scheme is hierarchical: keys matching the SAFE pattern are excluded from the credential inventory (they represent configuration values such as region identifiers, ARNs, or feature flags that do not require rotation). Keys matching the CRITICAL pattern are assigned P0 priority, indicating that they directly enable database access, authentication bypass, or payment system abuse. HIGH-priority keys (P1) represent third-party service credentials whose compromise enables data exfiltration or service abuse. Unrecognised keys that do not match any pattern are assigned REVIEW status for manual examination.

3 Rotation Priority Matrix

The classification output feeds into a rotation priority matrix. Table 1 presents the severity tiers, associated key patterns, representative examples, and rotation urgency applied in the incident analysed in this paper.

Table 1

Lambda credential rotation priority matrix

Priority	Key Pattern	Example	Rotation Window
P0 Critical	*password*, *passwd*, *jwt*, *secret_key*, *stripe_secret*, *db_pass*, *database_url*	DB_PASSWORD, JWT_SECRET, STRIPE_SECRET_KEY	Immediate (24h)
P1 High	*api_key*, *token*, *auth*, *firebase_key*, *amplitude*, *zendesk*, *hmac*	AMPLITUDE_API_KEY, ZENDESK_TOKEN, FIREBASE_PRIVATE_KEY	Within 7 days
P2 Medium	*webhook_secret*, *signing_secret*, *basic_auth*, *admin_pass*	WEBHOOK_SECRET, SWAGGER_BASIC_AUTH	Within 30 days
P3/Safe	*_arn*, *_url*, *_name*, *_id*, *stage*, *region*, *log_level*, *feature_*	S3_BUCKET_NAME, STAGE, LOG_LEVEL, REGION	No rotation required

In the analysed incident, P0 credentials included the production database password, JWT signing secrets, and Stripe payment API keys. P1 credentials spanned Amplitude analytics, Firebase mobile backend, Reteno CRM, and Zendesk customer support integrations. The total credential surface comprised 47 distinct secrets across ~1000 functions, with significant duplication - the same database password appeared as an environment variable in 89 separate functions.

ROTATION AND MIGRATION FRAMEWORK

1 Immediate Rotation Procedure

Credential rotation must be coordinated across three dimensions simultaneously: the source system (where the credential authenticates), the AWS Lambda environment variable, and any other consumers of the credential (CI/CD pipelines, mobile application configuration, RDS Proxy). Uncoordinated rotation - rotating the source without updating Lambda - results in service outages. The following procedure minimises downtime for database credential rotation, which represents the highest-risk and most architecturally complex case:

First, a new secret version is created in AWS Secrets Manager using the built-in Single User rotation Lambda rotator. Secrets Manager creates the new password, updates the secret, and maintains an AWSPREVIOUS version for graceful rollover. Second, Lambda functions are updated in parallel to reference the new credential, either by updating the environment variable directly or, in the target architecture, by updating the Secrets Manager secret ARN reference. Third, the RDS Proxy - which mediates connections between Lambda and RDS - is configured to use Secrets Manager authentication, enabling it to automatically pick up rotated credentials without Lambda restarts. Fourth, CloudWatch

metrics (DatabaseConnections, LoginFailures) are monitored for 15 minutes post-rotation to confirm successful credential propagation.

2 Migration to AWS Secrets Manager

Rotation addresses the immediate incident but does not resolve the underlying vulnerability: as long as credentials are stored in Lambda environment variables, any future GetFunction call from a sufficiently privileged identity will expose them in plaintext (Datadog Security Labs, 2024). The definitive remediation is the migration of all secrets from Lambda environment variables to AWS Secrets Manager with runtime fetch (Amazon Web Services, 2024).

The target architecture replaces inline environment variable storage with a Secrets Manager call executed at Lambda cold start, outside the handler function. The fetched secret is cached in the execution environment for the container lifetime, eliminating per-invocation latency overhead. The Lambda execution role is granted a least-privilege IAM policy restricted to GetSecretValue on the specific secret ARN:

```
// Node.js - cold start secret fetch pattern
import { SecretsManagerClient,
         GetSecretValueCommand } from '@aws-sdk/client-secrets-manager';

const sm = new SecretsManagerClient({});
let cachedSecret;

async function getSecret() {
  if (cachedSecret) return cachedSecret;
  const { SecretString } = await sm.send(
    new GetSecretValueCommand({ SecretId: 'prod/app/db' })
  );
  cachedSecret = JSON.parse(SecretString);
  return cachedSecret;
}

export const handler = async (event) => {
  const { DB_PASSWORD } = await getSecret();
  // use DB_PASSWORD for database connection
}
```

In this architecture, a GetFunction call returns an empty Variables object. The credential is never stored in Lambda configuration and cannot be exfiltrated through the Lambda API surface. The attack vector described in this paper is completely eliminated.

3 IAM Hardening

In addition to migration, access to the GetFunction API should be restricted at the IAM policy level (Shields, 2022). A deny statement added to Lambda execution roles prevents the roles themselves from calling GetFunction, limiting the API to explicitly authorised CI/CD service accounts and operations roles. A Service Control Policy applied at the AWS Organizations level can enforce this restriction account-wide:

```
{
  "Effect": "Deny",
  "Action": "lambda:GetFunction",
  "Resource": "*",
  "Condition": {
    "StringNotLike": {
      "aws:PrincipalArn": [
        "arn:aws:iam::*:role/DevOpsDeploymentRole",
        "arn:aws:iam::*:role/SecurityAuditRole"
      ]
    }
  }
}
```

PREVENTIVE CONTROLS AND DETECTION

The incident described in this paper could have been prevented or detected at multiple layers (Morrow, 2021). This section summarises the control categories and their effectiveness based on incident reconstruction.

AWS GuardDuty would have generated an UnauthorizedAccess:IAMUser/InstanceCredentialExfiltration finding within minutes of the attacker's first API calls from the IP address. GuardDuty's machine learning models for credential abuse pattern detection are specifically designed for this class of attack. The absence of GuardDuty was the single most impactful missing control in this incident.

AWS Config managed rule lambda-no-plaintext-envvars provides continuous compliance monitoring, alerting when Lambda functions have credentials stored in environment variables. This rule operates independently of whether an active attack is underway, providing proactive identification of functions that represent latent risk.

Pre-commit secret scanning using TruffleHog or Gitleaks, integrated into the CI/CD pipeline as a mandatory gate, would have prevented the initial credential exposure that enabled the attack. TruffleHog, the same tool used by the attacker to discover the exposed root key, can detect over 700 credential types across all common development artefact types including Git history, CI/CD logs, and container registries (TruffleHog, 2024).

CloudWatch metric alarms on GetSecretValue, CreateAccessKey, and GetFunction API calls, combined with CloudTrail-based metric filters, provide near-real-time alerting on credential access patterns. The critical alarm is on UpdateAccountSendingEnabled (SES), which in this incident enabled a phishing campaign before the Lambda exfiltration was detected. A complete alerting framework should monitor all high-risk API calls listed in AWS Security Best Practices documentation (Shields, 2022; Amazon Web Services, 2024).

CONCLUSION

This paper has presented a comprehensive analysis of the Lambda environment variable credential exfiltration threat, grounded in a real-world incident affecting a production AWS environment. The incident demonstrates that the convenience of Lambda environment variable storage for credentials creates a systemic architectural vulnerability: a single compromised high-privilege credential provides access to the entire credential surface of all deployed Lambda functions through the standard GetFunction API.

The forensic methodology presented - CloudTrail extraction, automated classification using pattern-based regular expressions, and prioritised rotation tracking - provides a reproducible framework for incident response teams investigating similar events. The Python toolchain described is designed to operate without exposing secret values, enabling safe execution in shared environments including AWS CloudShell.

The definitive architectural remediation - migration from Lambda environment variables to AWS Secrets Manager with cold-start runtime fetch - eliminates the vulnerability class entirely while imposing minimal operational overhead through execution environment-level caching. Organisations that complete this migration are resilient against the attack vector regardless of the privilege level of any future compromised credential.

Future work should address three concrete directions. First, automated compliance validation: an AWS Config custom rule set should enforce that Lambda functions in target accounts contain empty Variables objects, with a remediation workflow that flags non-compliant functions, routes notifications through Security Hub, and triggers an automated migration pipeline. This would provide continuous, measurable evidence of migration completeness rather than a point-in-time audit. Second, runtime secret monitoring: a Lambda Extension implementing a credential access telemetry agent should be developed and validated. Such an extension could intercept Secrets Manager API calls at the execution environment layer, record access metadata (SecretId, caller identity, timestamp) to a centralised observability platform, and generate anomaly alerts when a secret is accessed from an unexpected function identity or geographic region - providing a detection layer that operates independently of IAM-level controls. Third, defence and governmental applicability: the incident taxonomy and toolchain developed in this paper should be extended to address multi-account AWS Organizations deployments typical of governmental and military cloud architectures, where the credential exfiltration attack surface spans organisational units and cross-account trust boundaries. Automated tooling for cross-account Lambda credential inventories, combined with Security Control Policy enforcement at the organisations level, would address the systemic risk in such environments.

References

1 Amazon Web Services. (2024) AWS Lambda Developer Guide. Available at: https://docs.aws.amazon.com/lambda/latest/dg/

2 Amazon Web Services. (2024) AWS Secrets Manager User Guide. Available at: https://docs.aws.amazon.com/secretsmanager/latest/userguide/

3 Amazon Web Services. (2023) CloudTrail Log Event Reference. Available at: https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-event-reference.html

4 Brewer, R. (2014) 'Advanced persistent threats: minimising the damage', Network Security, 2014(4), pp. 5-9.

5 Dua, S. and Du, X. (2016) Data Mining and Machine Learning in Cybersecurity. Boca Raton: CRC Press.

6 Marin, E., Perino, D. and Di Pietro, R. (2022) 'Serverless computing: a security perspective', Journal of Cloud Computing, 11(1), pp. 1-12.

7 Hutchins, E.M., Cloppert, M.J. and Amin, R.M. (2011) 'Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains', Leading Issues in Information Warfare and Security Research, 1(1), pp. 80-106.

8 MITRE Corporation. (2024) ATT&CK for Cloud: T1552.001 - Unsecured Credentials: Credentials In Files. Available at: https://attack.mitre.org/techniques/T1552/001/

9 Morrow, T. (2021) '12 risks, threats, and vulnerabilities in moving to the cloud', Software Engineering Institute Blog. Available at: https://insights.sei.cmu.edu/blog/12-risks-threats-vulnerabilities-in-moving-to-the-cloud/

10 Datadog Security Labs. (2024) 'Secrets exposed in Lambda function environment variables', Cloud Security Atlas. Available at: https://securitylabs.datadoghq.com/cloud-security-atlas/vulnerabilities/lambda-function-secrets-in-environment-variables/

11 Rid, T. and Buchanan, B. (2015) 'Attributing cyber attacks', Journal of Strategic Studies, 38(1-2), pp. 4-37.

12 TruffleHog. (2024) TruffleHog: Find leaked credentials. Available at: https://github.com/trufflesecurity/trufflehog

13 Shields, D. (2022) AWS Security. Shelter Island: Manning Publications.