RESEARCH ARTICLE OPEN ACCESS

Enhancing AWS Cognito Authentication with Pre-Authentication Lambda Triggers for Group-Based Access Control and 2FA Integration

Sangeeta Manna,
BSc. Hons. (Cal.), Master of Computer Applications (UTech. WB. India)
Consultant Software Programmer in Banking, Finance & Service-related IT Industry
India

ABSTRACT
This paper presents a real-world implementation and analysis of AWS Cognito's Pre-Authentication Lambda trigger to enforce group-based access control before user login and to seamlessly integrate Multi-Factor Authentication (2FA). It also identifies common pitfalls in Lambda output handling that disrupt the Cognito login flow and provides a working solution validated with a frontend-initiated login request. The findings serve as a practical guide for developers and architects aiming to implement secure, group-aware authentication flows with AWS Cognito.

Keywords: AWS Cognito (Amazon Web Services Cognito), Lambda Trigger, Pre-Authentication, MFA (Multi-Factor Authentication), 2FA (Two Factor Authentication), One time password (OTP), Group-Based Access Control (GBAC), Identity and Access Management (IAM), Cloud Security, JavaScript Object Notation (JSON).

I. INTRODUCTION

AWS Cognito [1] is a widely used user identity service that enables user authentication, authorization, and access control for web and mobile applications. Developers often face challenges in correctly validating users based on custom logic such as group membership prior to authentication. AWS Cognito [1] has default behaviour using Lambda triggers [4], which allow pre- and post-authentication workflows. However, many organizations require group-based validation before a user is authenticated—ensuring that only users from specific roles or portals can log in. This research focuses on leveraging Cognito's Pre-Authentication trigger to validate user groups membership dynamically and integrate with Cognito's native 2FA [3] capabilities, ensuring compliance and security in multi-tenant or role-based environments.

II. PROBLEM STATEMENT

The primary challenge addressed in this research is to enable Amazon Cognito to enforce user group validation through a Pre-Authentication Lambda trigger [4], while also maintaining seamless compatibility with 2FA [3] workflows based on one-time passwords (OTP). Due to Cognito's stringent requirements for Lambda responses, improper handling of the event object often results in unexpected authentication failures, such as the "*InvalidLambdaResponseException*." Therefore, the proposed solution must ensure that the validation logic integrates harmoniously with Cognito's authentication lifecycle, preserving both security and functional continuity, while enabling robust, group-based access control.

III. AIMS AND OBJECTIVES

The primary aim of this research is to design and implement a secure, scalable authentication mechanism that enables group-based access control within Amazon Cognito, while maintaining compatibility with its MFA [3] process. The study specifically focuses on leveraging Cognito's Pre-Authentication Lambda trigger to validate user groups dynamically before allowing access, without disrupting Cognito's built-in authentication lifecycle.

The specific objectives of this research are to:

1. Implement group-based access control using Cognito's Pre-Authentication Lambda trigger — by validating user groups at runtime and ensuring that users belong to the

correct Cognito group before authentication proceeds.

2. Ensure smooth integration with Cognito's MFA (2FA) [3] flow — by designing the Lambda logic to coexist seamlessly with Cognito's OTP-based verification process, without causing interruptions or invalid response errors.

3. Identify common pitfalls in Lambda event return handling and propose reliable solutions — through detailed analysis of error cases such as "*InvalidLambdaResponseException*", and by establishing a best-practice pattern for handling and returning event objects properly.

4. Demonstrate a working and validated architecture using frontend-initiated login requests — by implementing a practical prototype where user authentication requests from a frontend client (such as a web application) successfully trigger Cognito's workflow, apply group validation, and complete MFA-based login securely.

IV. METHODOLOGY

The methodology of this research focuses on integrating group-based access control into Amazon Cognito's authentication framework using a Pre-Authentication Lambda trigger [4], while ensuring smooth compatibility with multi-factor authentication (MFA [3]). The implementation follows a structured sequence of operations across the frontend, Cognito, and Lambda layers, designed to maintain both security and functional consistency.

A. System Architecture Overview

The overall architecture consists of four key components that interact during the authentication lifecycle:

1. Frontend (Client Application)
    - Initiates user authentication using the USER_PASSWORD_AUTH flow.
    - Sends the authentication request to Cognito, including credentials (username, password) and contextual parameters such as *portalType* in the *ClientMetadata* field.
    - This metadata enables Cognito and the Lambda trigger [4] to determine which portal (e.g., Admin, User, Business-User) the user is trying to access.
2. Cognito User Pool
    - Serves as the identity provider and orchestrates the authentication workflow.
    - Before validating the user credentials, Cognito automatically triggers the Pre-Authentication Lambda [5] [7] function.
    - The Lambda function determines whether the login request complies with business rules—specifically, whether the user belongs to the required group for the selected portal type.
3. Pre-Authentication Lambda Trigger (Java Implementation)
    - Executes custom validation logic to ensure that only users from appropriate Cognito groups can authenticate through specific portals.
    - Retrieves parameters such as *portalType* and user details from the event object.
    - Returns the original event object to Cognito if validation passes;

otherwise, throws an exception to block authentication.

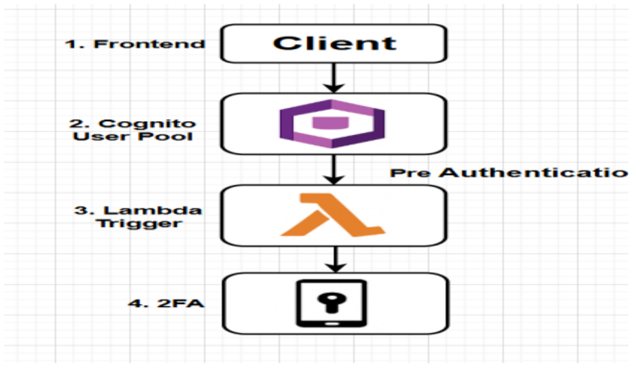
- The strict requirement to return the event object ensures that Cognito's lifecycle continues normally and MFA [3] remains functional.

4. MFA (2FA) Integration

- If MFA is enabled in the user pool, Cognito automatically initiates OTP-based verification after the Lambda trigger [4] completes successfully.

- Since the Lambda returns the event correctly, Cognito continues seamlessly to the MFA step without errors like *InvalidLambdaResponseException*.

B. Authentication Flow Diagram



The interaction between components can be represented as a five-step logical data flow:

Step 1: Frontend → Cognito

- The frontend sends an authentication request using the InitiateAuth API with AuthFlow: USER_PASSWORD_AUTH. Initially there is no interaction with backend service. Frontend generate request using sdk.

- Request payload includes credentials and ClientMetadata (e.g., portalType = ADMIN).

Step 2: Cognito → Lambda (PreAuthentication Trigger [5] [7])

- Cognito intercepts the request and invokes the configured Pre-Authentication Lambda function before validating credentials.

- The Lambda receives the event object containing userName, userAttributes, and clientMetadata.

Step 3: Lambda Validation Logic

- Lambda checks the user's group membership (retrieved from Cognito) against the expected group for the provided portalType.

- If validation passes → return event;

- If validation fails → throw new *RuntimeException*("User not authorized for this portal.");

Step 4: Cognito → MFA Workflow

- Upon receiving the returned event, Cognito proceeds with its built-in authentication lifecycle.

- If MFA is enabled, Cognito triggers [5] OTP verification automatically. OTP will be sent as per user preference (either email or Mobile).

Step 5: Cognito → Frontend (Token Issuance)

- Once MFA is verified, Cognito issues ID, Access Token, and Refresh Tokens to the frontend.

- The user is now authenticated with enforced group-based access control.

C. Frontend Request Example

Below is the sample payload structure used in the frontend to initiate the authentication request:

```
{ "AuthFlow": "USER_PASS_AUTH", "ClientId": "<client_id>", "AuthParameters": { "USERNAME": "<username>", "PASSWORD": "<password>", "SECRET_HASH": "<secret_hash>" }, "ClientMetadata": { "portalType": "ADMIN" } }
```

- AuthFlow: Defines the authentication mechanism [8].
- ClientId: Uniquely identifies the Cognito client app.
- AuthParameters: Contains user credentials and security information.
- ClientMetadata: Provides additional validation context (e.g., portal type).

D. Lambda Implementation Example (Java)

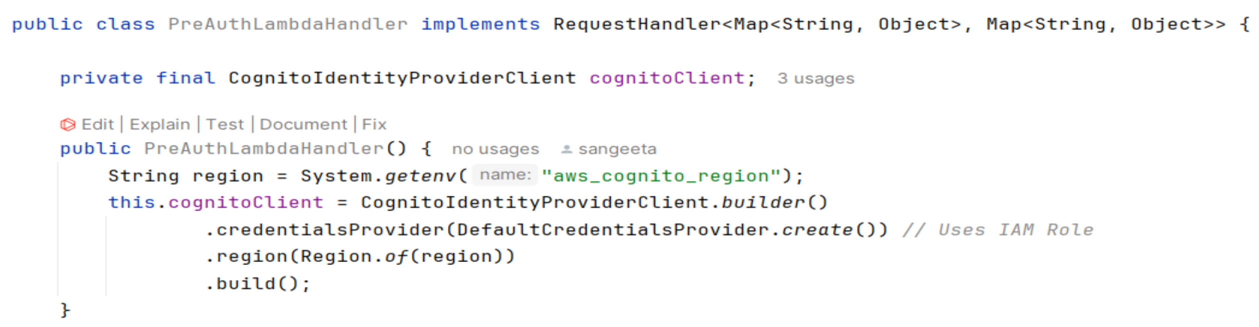
A simplified version of the Pre-Authentication Lambda logic is as follows:

```
public Map<String, Object> handleRequest(Map<String, Object> event,
Context context) {
    boolean isValidUserGroup = validateUserGroup(event);
    if (isValidUserGroup) {
        return event; // Mandatory for successful authentication
    } else {
        throw new RuntimeException("User does not belong to the required
group.");
    }
}
```

Key Considerations:

- Always return the original event object for Cognito to continue authentication.
- Throwing an exception immediately stops the process and denies access.
- Use clientMetadata (e.g., portalType) and Cognito Admin APIs (like AdminListGroupsForUser) for group validation.

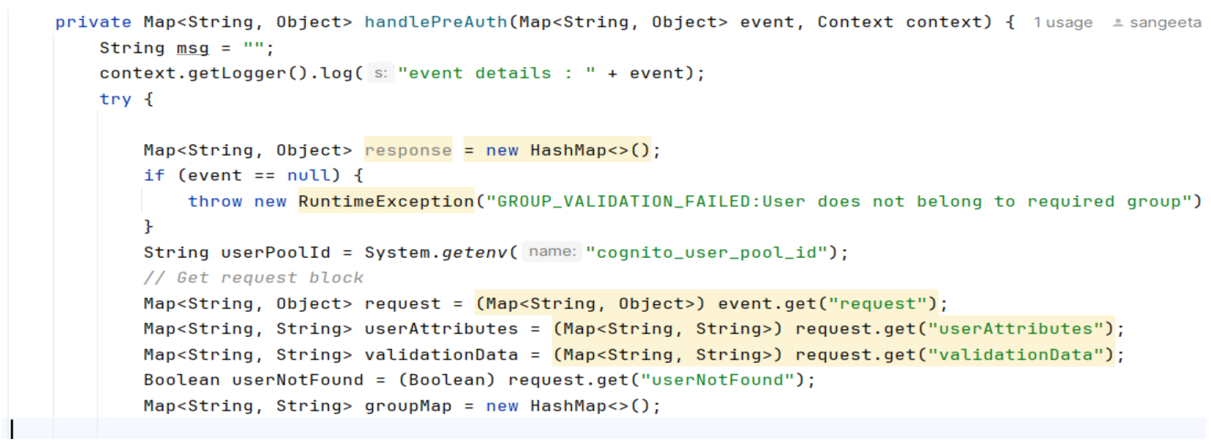
Sample Code:



```
public class PreAuthLambdaHandler implements RequestHandler<Map<String, Object>, Map<String, Object>> {

    private final CognitoIdentityProviderClient cognitoClient;

    public PreAuthLambdaHandler() {
        String region = System.getenv("aws_cognito_region");
        this.cognitoClient = CognitoIdentityProviderClient.builder()
                .credentialsProvider(DefaultCredentialsProvider.create()) // Uses IAM Role
                .region(Region.of(region))
                .build();
    }
```



```
private Map<String, Object> handlePreAuth(Map<String, Object> event, Context context) {  1 usage  sangeeta
    String msg = "";
    context.getLogger().log( s: "event details : " + event);
    try {

        Map<String, Object> response = new HashMap<>();
        if (event == null) {
            throw new RuntimeException("GROUP_VALIDATION_FAILED:User does not belong to required group")
        }
        String userPoolId = System.getenv( name: "cognito_user_pool_id");
        // Get request block
        Map<String, Object> request = (Map<String, Object>) event.get("request");
        Map<String, String> userAttributes = (Map<String, String>) request.get("userAttributes");
        Map<String, String> validationData = (Map<String, String>) request.get("validationData");
        Boolean userNotFound = (Boolean) request.get("userNotFound");
        Map<String, String> groupMap = new HashMap<>();
```

Sample Log from AWS CloudWatch:

```
START RequestId: xxxxxx-xxxxxx-xxxxxx-xxxxx Version: $LATEST

Before Trigger source: PreAuthentication_Authentication

Check data inside event before Trigger source:
event details : {version=1, region=xxxxx, userPoolId=xxxxxx, userName=<user-name>, callerContext={awsSdkVersion=aws-sdk-js-3.726.0, clientId=yyyyyyyyyy}, triggerSource=PreAuthentication_Authentication, request={userAttributes={sub=xxxxxx-xxxxxx-xxxxx-xxxx, email_verified=true, cognito:user_status=CONFIRMED, phone_number_verified=false, phone_number=<phone_number>, email=<email_id>}, validationData={portalType=Business-User~Individual-User}, userNotFound=true}, response={}}

validationData: {portalType=Business-User~Individual-User}

Username, userFound value: <user-name>->true
User groups: [GroupType(GroupName=Business-User, UserPoolId=xxxxx-xxxxxx, Description=Group of Business users, LastModifiedDate=2025-06-10T13:04:37.570Z, CreationDate=2025-06-10T13:04:37.570Z)]
```

This event detail signifies that the user belongs to the Cognito User Pool with a confirmed status. The email is verified, while the phone number is not. In this scenario, if at least one attribute is verified, the Cognito Lambda will return the corresponding portalType. In this case, the user's portalType is Business-Use.

V. RELATED WORK & IMPLEMENTATION

Existing literature discusses general Lambda triggers, but few explore enforcing pre-auth group validation integrated with MFA. This implementation bridges that gap by combining group-based validation, error control, and secure MFA handoff in Cognito's authentication cycle. It also addresses JSON formatting and returns structure issues, providing tested reference code.

VI. RESULT ANALYSIS

- Valid User Group: Authentication proceeds normally, and Cognito triggers MFA instantly.

- Invalid Group: Lambda throws Runtime Exception, and Cognito halts login.
- Improper Lambda Return: Causes "*InvalidLambdaResponseException*" with messages like Invalid JSON or Invalid version in Lambda response.

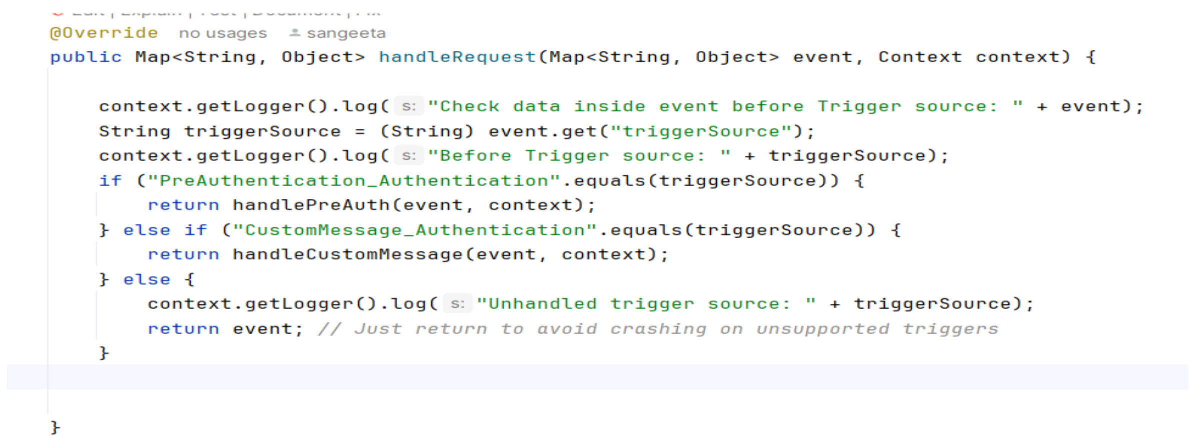
Expected Behavior Table:

Case	**Lambda Return Type**	**Cognito Result**
Valid group	Map<String, Object>	Authentication continues + 2FA
Invalid group	Exception	*RuntimeException*, Login blocked
Null/custom return	Error	*InvalidLambdaResponseException*

Error Handling Patterns

- User not in expected group: Throw *RuntimeException*
- Valid group and user: Return event map

Sample Code:



```
@Override   no usages   sangeeta
public Map<String, Object> handleRequest(Map<String, Object> event, Context context) {

    context.getLogger().log( s: "Check data inside event before Trigger source: " + event);
    String triggerSource = (String) event.get("triggerSource");
    context.getLogger().log( s: "Before Trigger source: " + triggerSource);
    if ("PreAuthentication_Authentication".equals(triggerSource)) {
        return handlePreAuth(event, context);
    } else if ("CustomMessage_Authentication".equals(triggerSource)) {
        return handleCustomMessage(event, context);
    } else {
        context.getLogger().log( s: "Unhandled trigger source: " + triggerSource);
        return event; // Just return to avoid crashing on unsupported triggers
    }


}
```

Sample log for error handling code from AWS CloudWatch:

Before Trigger source: PreAuthentication_Authentication

event details : {version=1, region=XXXXXX, userPoolId=XXXXXXX, userName=<user-name>, callerContext={awsSdkVersion=aws-sdk-js-3.726.0, clientId=YYYYYYYYYY}, triggerSource=PreAuthentication_Authentication, request={userAttributes={sub=xxxx-xxxx-xxxx, email_verified=false, cognito:user_status=UNCONFIRMED, phone_number_verified=false, phone_number=<phone-number>, email=<email-id>}, validationData={portalType=Business-User~Individual-User}, userNotFound=false}, response={}}

```
validationData: {portalType=Business-User~Individual-User}

Username, userNotFound value: <user-name>->false
```

Exception: Error: User is not in CONFIRMED status.

```
User is not in CONFIRMED status.: java.lang.RuntimeException

java.lang.RuntimeException: User is not in
CONFIRMED status. at
com.fusionfunds.echeck.preauth.lambda.cognito.Pre
AuthLambdaHandler.handlePreAuth(PreAuthLambd
aHandler.java:143) at
com.fusionfunds.echeck.preauth.lambda.cognito.Pre
AuthLambdaHandler.handleRequest(PreAuthLambda
Handler.java:42) at
java.base/jdk.internal.reflect.DirectMethodHandleAc
cessor.invoke(Unknown Source) at
java.base/java.lang.reflect.Method.invoke(Unknown
Source)
```

This event detail signifies that the user belongs to the Cognito User Pool but has a not confirmed status. Therefore, we cannot identify which group the user belongs to.

VII. FUTURE WORK

While this research successfully demonstrates the feasibility of integrating group-based access control with AWS Cognito's Pre-Authentication Lambda trigger, several enhancements can further strengthen the overall architecture and operational robustness. Future work will focus on the following key areas:

1. Implement Dynamic Group–Role Mappings:
   The current implementation validates user groups statically within Cognito. Future iterations should introduce dynamic group-to-role mappings, allowing flexible authorization policies based on external data sources or contextual attributes. This enhancement would enable fine-grained access control, adaptive to evolving business rules or user roles managed outside of Cognito.

2. Add Logging and Metrics for Auditing Purposes:
   To improve traceability and compliance, future development should include comprehensive logging and monitoring mechanisms. By integrating AWS CloudWatch or similar observability tools, system administrators can track authentication outcomes, group validation decisions, and trigger execution metrics. Such visibility will be valuable for auditing, performance optimization, and identifying potential security anomalies.

3. Extend to Post-Authentication for Session Enrichment:
   Beyond pre-authentication checks, extending the solution to Cognito's Post-Authentication trigger will allow for session enrichment—such as embedding user-specific claims, role attributes, or contextual metadata into the access token. This would enhance downstream authorization mechanisms and support more personalized, secure, and efficient session handling across services.

VIII. CONCLUSION

This work presents a consistent and secure approach to enhancing Amazon Cognito's authentication workflow through the use of the Pre-Authentication Lambda trigger [8]. The study highlights that strict adherence to Cognito's return contract—specifically returning the original event map—is crucial to maintaining compatibility with Cognito's internal authentication lifecycle. Failure to comply with this requirement often leads to errors such as *InvalidLambdaResponseException*, disrupting the login process.

By embedding group-based access control within the pre-authentication phase, the proposed solution effectively enforces user validation before authentication is finalized. Furthermore, the implementation ensures seamless interoperability with multi-factor authentication (MFA) or OTP-based flows, preserving both security and usability.

This work contributes a practical, validated architecture for integrating fine-grained access control in multi-portal systems that rely on Cognito, without requiring external identity orchestration. The proposed model not only strengthens the overall security posture but also lays the foundation for extending similar controls to post-authentication workflows and future enterprise-scale identity management solutions.

REFERENCES

[1] Amazon Web Services (AWS). "Amazon Cognito Developer Guide." AWS Documentation. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/

[2] Amazon Web Services (AWS). "Using AWS Lambda Triggers in Amazon Cognito User Pools." AWS Documentation. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-lambda-pre-authentication.html

[3] Amazon Web Services (AWS). "Multi-Factor Authentication (MFA) in Amazon Cognito." AWS Documentation. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/user-pool-settings-mfa.html

[4] AWS Security Blog. "Implementing fine-grained access control in Amazon Cognito using Lambda triggers." AWS Security Blog, 2023. Available: https://aws.amazon.com/blogs/security/

[5] AWS Architecture Blog. "Best practices for securing user authentication and access with Amazon Cognito." AWS Architecture Blog, 2022.

[6] Stack Overflow. "How to handle InvalidLambdaResponseException in Cognito PreAuthentication Lambda." Stack Overflow Thread, 2023.

[7] Medium. S. Patel, "Extending AWS Cognito with Lambda Triggers for Custom Authentication." Medium Blog, 2023.

[8] IEEE Access Journal. R. P. Singh, "Serverless Security Models: Leveraging AWS Cognito and Lambda for Scalable Authentication." IEEE Access, vol. 10, 2022.