# QATIP Intermediate
# AWS Lab 05
# Validations and Role-Based Access in AWS

## Contents

## Overview

This lab introduces Terraform validation mechanisms (preconditions and postconditions) and enforces secure, least-privilege IAM role usage through AWS AssumeRole mechanisms. You'll switch between users, roles, and credentials to explore how access is controlled and verified during resource deployment.

## Before you begin

1. Ensure you have completed Lab0 before attempting this lab.
2. In the IDE terminal pane, enter the following command...

   **cd ~/environment/aws-tf-int/labs/05**

3. This shifts your current working directory to labs/05. Ensure all commands are executed in this directory
4. Close any open files and use the Explorer pane to navigate to and open the labs/05 folder.

## Phase 0 – Setup: Creating a Limited Test User and IAM Role

### Purpose

- Prepare the environment by disabling Cloud9's temporary credentials and establishing a realistic least-privilege scenario.
- Create a new IAM user, testuser, and define a role it can assume, with limited permissions.

## Steps Summary

- Disable managed credentials in Cloud9.
- Configure AWS CLI using awsstudent credentials.
- Create testuser and assign a password (for console access if needed).
- Create a role TerraformLimitedAccessRole with permissions limited to EC2 and S3.
- Define a trust policy to allow testuser to assume this role.
- Attach an assume-role policy to testuser.

## Rationale

- By creating a constrained test environment, this phase simulates real-world identity-based access where users can't directly perform privileged actions but must assume authorized roles.

## Steps

1. Remove current credentials:

   **rm -f ~/.aws/credentials**

2. Go to AWS Cloud9 → Preferences → AWS Settings → Credentials and set/verify "AWS managed temporary credentials" to Off.

3. Configure AWS CLI to use 'awsstudent' credentials:

   **aws configure**

   AWS Access Key ID: **(your awsstudent key)**
   AWS Secret Access Key: **(your awsstudent secret key)**
   Region: **us-east-1**
   Output format: (leave blank)

4. Create a new IAM user called 'testuser':

   **aws iam create-user --user-name testuser**

5. Grant console access — not strictly required for Terraform

   **aws iam create-login-profile --user-name testuser **
   **--password 'TestPassword123!'**

6. Review the trust policy file **trust-policy.json** and update line **7**, replacing **<Account_ID>** with your lab account ID.
7. Create a new IAM role '**TerraformLimitedAccessRole**' that grants access to EC2 and S3 using the **trust-policy.json** file:

```
aws iam create-role --role-name TerraformLimitedAccessRole \
 --assume-role-policy-document file://trust-policy.json
```

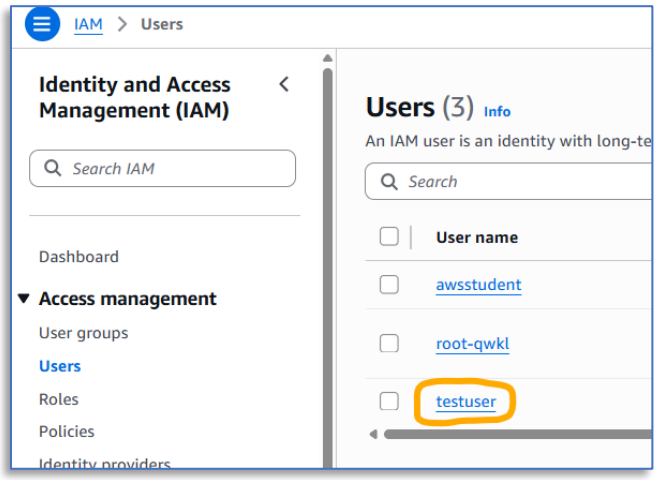
8. Review the provided inline policy for this role, **ec2-s3-policy.json**, and then attach it to the role:

```
aws iam put-role-policy --role-name TerraformLimitedAccessRole \
--policy-name EC2andS3Permissions \
--policy-document file://ec2-s3-policy.json
```

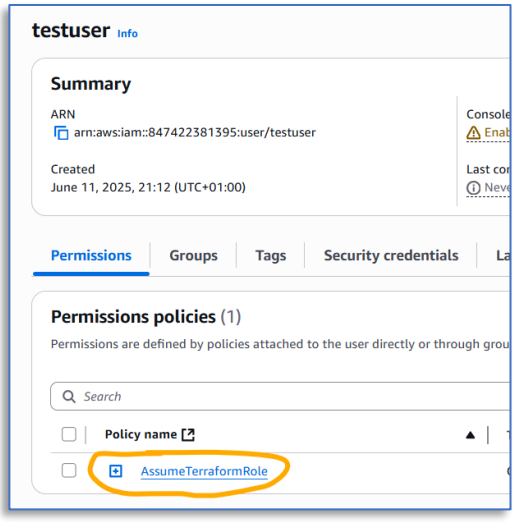
9. Review the assume role policy file **assume-role-policy.json** and update line **6**, replacing **<Account_ID>** with your lab account ID.
10. Attach the policy to 'testuser' using **assume-role-policy.json:**

```
aws iam put-user-policy --user-name testuser \
 --policy-name AssumeTerraformRole \
--policy-document file://assume-role-policy.json
```

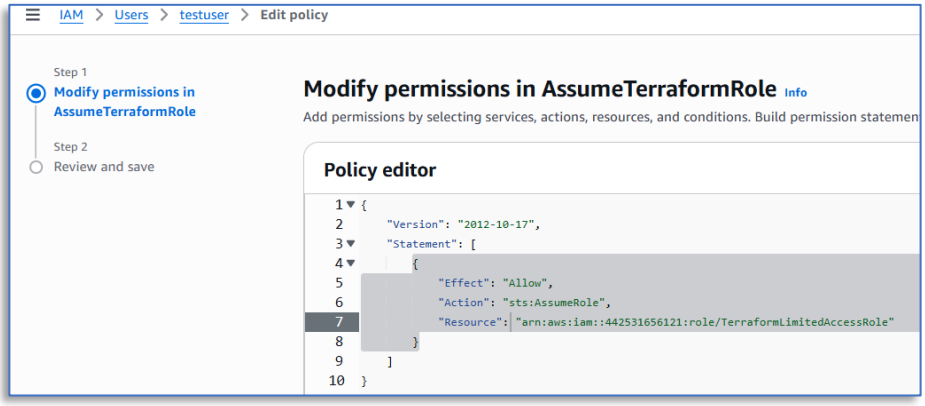
11. Switch to the console to verify the current configuration..
    a. The user testuser is created...



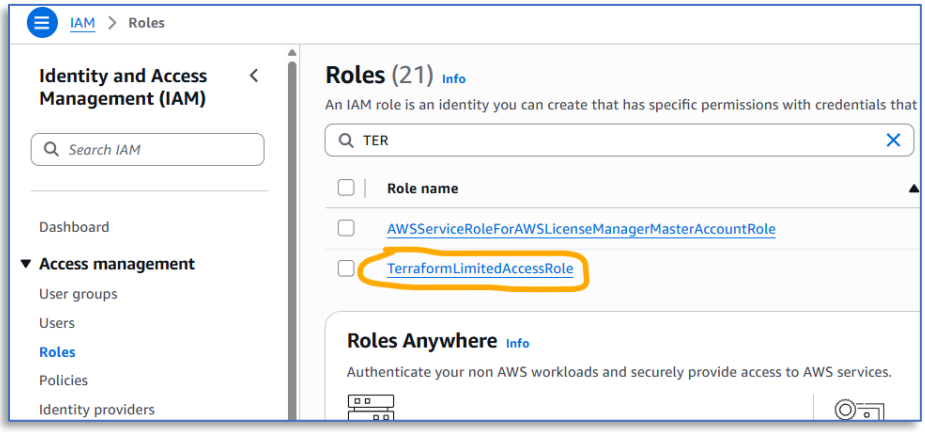

    b. The testuser has policy “AssumeTerraformRole” attached...



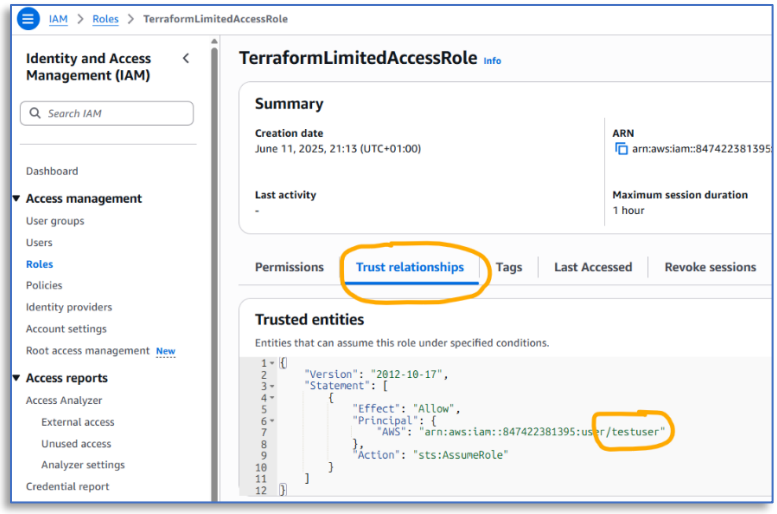

c. The AssumeTerraformRole policy grants testuser the permission to assume a role...



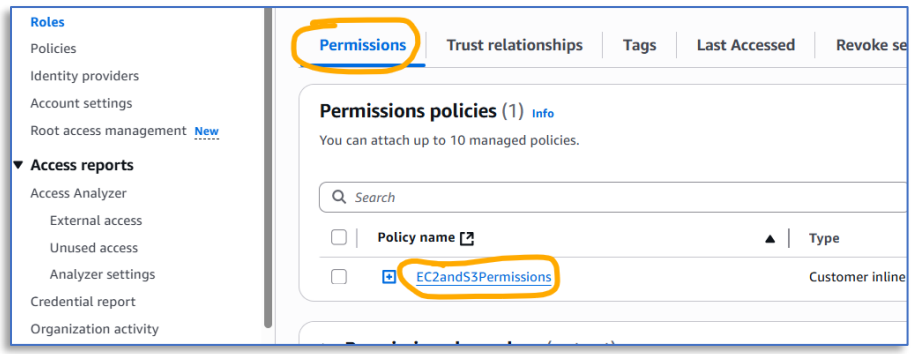

d. A role named “TerraformLimitedAccess” is created...



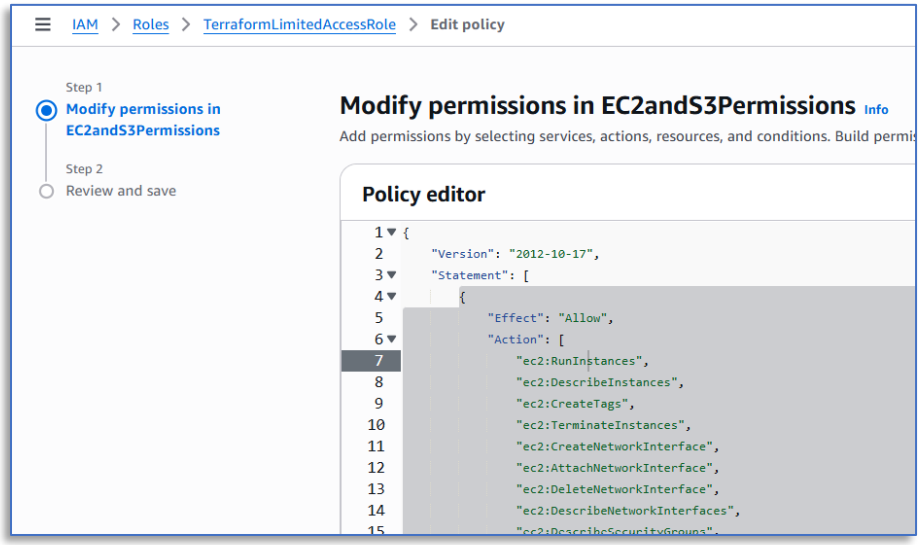

e. The user testuser can assume this role...




f. The role has permissions policy “EC2andS3Permissions” attached …




g. The policy “EC2andS3Permissions” gives the user of the role permissions in EC2 and S3…



## Phase 1 – Simulate the Testuser Identity

### Purpose

- Temporarily adopt the identity of the test user to experience the access restrictions firsthand.

### Steps Summary

- Generate and configure access keys for testuser.
- Confirm identity with aws sts get-caller-identity.

### Rationale

- This phase ensures Terraform will operate under the limited user context, which becomes important for demonstrating permission denials and later, proper role assumption.

### Steps

1. Generate access keys for 'testuser':

   **aws iam create-access-key --user-name testuser**

2. Note down the AccessKeyId and SecretAccessKey
3. Configure AWS CLI to use testuser credentials:

   **aws configure**

   AWS Access Key ID: **(testuser Access key)**
   AWS Secret Access Key: **(testuser Secret key)**
   Region: **us-east-1**
   Output format: (leave blank)

4. Verify these are your current credentials:

   **aws sts get-caller-identity**

5. Terraform will now run under this identity

## Phase 2 - Terraform Without Assuming the Role

### Purpose

- Observe what happens when a user attempts to deploy resources without the appropriate permissions.

### Steps Summary

- Populate terraform.tfvars with randomized identifiers.
- Run terraform init and terraform apply as testuser.
- Expect permission errors for EC2 and S3 operations.
- Investigate the main.tf file and uncomment the role block to enable AssumeRole.
- Fix missing permissions (e.g., ec2:DescribeInstanceCreditSpecifications) by reapplying the policy (requires switching back to awsstudent).

### Rationale

This phase highlights how assuming roles is required for privilege escalation and how policy updates must be done from a privileged account.

### Steps

1. Update terraform.tfvars, replacing <random-4-digits> with 4 random digits.
2. Run **terraform init**
3. Run **terraform plan** – planning should succeed
4. Run **terraform apply** - 2 permissions errors should be generated (ec2:RunInstances and s3:CreateBucket)
5. Review **ec2-s3-policy.json** and verify that those permissions are granted. The issue here is that the testuser has not assumed the role to which the permissions apply
6. Uncomment lines 5 through 9 of **main.tf** and update line **6** with your account ID
7. Run **terraform apply** – This generates new permission errors. Switch the GUI and verify that the 2 resources are created. The permissions error relate to getting information about the newly created resource. Failing to retrieve this information means the statefile is not accurate.

8. Update the file **ec2-s3-policy.json** file to include the 2 missing permissions (**ec2:DescribeInstanceCreditSpecifications** and **s3:GetBucketObjectLockConfiguration**) and then attempt to reapply the aws iam put-role-policy command:

   **aws iam put-role-policy --role-name TerraformLimitedAccessRole **
   **--policy-name EC2andS3Permissions **
   **--policy-document file://ec2-s3-policy.json**

9. An error is generated because testuser does not have the **iam:PutRolePolicy** permission, needed to update the policy. To fix this you will need to switch back to using your elevated awsstudent account. In production, this would be an escalation issue.
10. Use aws configure to switch to using awsstudent credentials

    **aws configure**

    AWS Access Key ID: (**awsstudent Access key**)
    AWS Secret Access Key: (**awsstudent Secret key**)
    Region: **us-east-1**
    Output format: (leave blank)

11. Re-run the aws iam put-role-policy command:

    **aws iam put-role-policy --role-name TerraformLimitedAccessRole **
    **--policy-name EC2andS3Permissions **
    **--policy-document file://ec2-s3-policy.json**

12. Use aws configure to switch back to using testuser credentials

    **aws configure**

    AWS Access Key ID: (**testuser Access key**)
    AWS Secret Access Key: (**testuser Secret key**)
    Region: **us-east-1**
    Output format: (leave blank)

13. Run **terraform apply**. The deployment now progresses successfully with the old resources being destroyed before new copies are created.

14. Tidy up with **terraform destroy**

## Phase 3 – EC2 Instance Type Precondition Check

### Purpose

- Demonstrate how Terraform can enforce guardrails to prevent undesired configurations.

### Steps Summary

- Modify instance type to m5.large in terraform.tfvars.
- Run terraform plan — observe failure due to precondition.
- Revert to t2.small — observe success.

### Rationale

- Preconditions help enforce organizational standards before provisioning happens. In this case, instance types not beginning with t2 are disallowed.

### Steps

1. Update **terraform.tfvars** to attempt to provision an EC2 instance of type 'm5.large'.
2. Run **terraform plan**
3. Terraform planning fails with a validation error (precondition). The precondition checks that the variable **var.instance_type** begins with "t2"
4. Update terraform.tfvars, reverting the instance type to **“t2.small”**
5. Run **terraform plan**
6. The planning is successful as the variable **var.instance_type** now meets the precondition.

# Phase 4 – Bucket Name Validations (Precondition and Postcondition)

## Purpose

- Explore the difference between validating input variables (precondition) and verifying actual outcomes (postcondition) after logic is applied.

## Steps Summary

- Attempt to create an S3 bucket that violates a precondition.
- Modify the bucket parameters to meet the precondition.
- Introduce a logic error to invoke a postcondition failure.
- Undo the error to meet the postcondition.
- Apply the deployment and verify resources are created.
- Clean up by destroying resources.

## Rationale

- Preconditions catch bad input early; postconditions ensure that no unexpected transformations occur during deployment logic. This phase simulates bugs or misconfigurations that can pass initial validation but result in unintended infrastructure. Postconditions help catch these issues.

## Steps

1. Run **terraform plan**
2. Planning should be completed without errors as all preconditions and postconditions are currently met.
3. Update **terraform.tfvars**, changing the **bucket_name** variable value from "lab-bucket-xxxx" to "my-bucket-xxxx"
4. Run **terraform plan**
5. Terraform planning fails with a validation error (precondition). The precondition checks that the variable **var.bucket_name** contains "lab-bucket"

6. Undo the changes to terraform.tfvars, reverting the bucket_name variable back to "lab-bucket-xxxx"
7. Run **terraform plan**
8. The planning should be successful as the preconditions are now met
9. You will now introduce a logic error that will be captured by a postcondition check
10. Modify line **28** of main.tf, replacing the 2nd instance of "**lab-bucket**" with "**test-bucket**". This is to simulate a logic error whereby the bucket name is changed from the value supplied as a variable (which passes the precondition check) resulting in the actual bucket name being something other than intended. The postcondition check validates the actual name that would be given to the bucket once all terraform logic is applied.
11. Run **terraform plan**
12. Terraform fails with a validation error (postcondition).
13. Revert the changes to line **28**, re-aligning the actual bucket name back to the name derived from the variable.
14. Run **terraform plan**
15. Verify successful deployment with **terraform apply**
16. Switch to UI to verify EC2 instance and S3 bucket are created
17. Clean up with **terraform destroy**
18. Revert to using your **awsstudent** credentials in Cloud9 using **aws configure** again.

# Lab Summary Reflection: Validations and Role-Based Access in AWS

In this lab, you explored key DevOps principles around **security, access control, and configuration validation** in a real-world AWS Terraform deployment context.

## What You Demonstrated

**Secure Identity Management:**
You created a least-privilege IAM user and configured Terraform to assume a tightly scoped role. This mirrors best practices in enterprise environments where users never have direct access to sensitive resources.

**AssumeRole in Action:**
You experienced firsthand how Terraform fails without role assumption — reinforcing the principle of least privilege and the need for controlled privilege elevation.

**IAM Policy Debugging:**
You met and resolved permission errors by updating IAM policies. This proved how fine-grained permissions often require iterative tuning and the importance of auditing roles vs. users.

**Terraform Preconditions and Postconditions:**
You implemented both:

- **Preconditions** to confirm user input and enforce naming/instance standards.
- **Postconditions** to catch unexpected results or logic errors that could lead to misconfigured infrastructure.

## Why This Matters

In production environments:

- Misapplied permissions can expose resources or break automation.
- Terraform plans may appear valid but still deploy unintended resources due to logic flaws.

- Role-based access enables secure collaboration across teams, while validation provides safety nets against human error.
- This lab hopefully helped build your confidence in:
    - Debugging IAM-related failures
    - Structuring secure Terraform deployments.
    - Writing and understanding Terraform validation blocks.

## Reflection Questions

What would happen if you were to run this lab with full admin privileges instead of a limited user?

How would you extend these validation patterns to other resources like RDS, VPCs, or Lambda functions?

How can you apply the assume-role pattern to CI/CD pipelines securely?