QATIP Intermediate
Azure Lab05
Role-Based Access and Terraform Validations

Contents

Overview

This lab introduces Terraform validation mechanisms (preconditions and postconditions) and enforces secure, least-privilege access using **Azure Service Principals (SPs)**. You will:

- Create a Service Principal with limited permissions scoped to the subscription
- Use SP credentials via environment variables
- Run Terraform operations and observe RBAC restrictions
- Upgrade permissions and reattempt deployments
- Implement Terraform validations

Before You Begin

1. Ensure you have Owner or User Access Administrator permissions at the **subscription** level.
2. Azure CLI and Terraform must be installed.
3. Navigate to the lab directory:

   cd c:\azure-tf-int\labs\05

Phase 0 – Create a Limited-Service Principal and Set Environment Variables

Purpose

What we are doing:
We are creating a new **non-human identity** (Service Principal) that will be used to run Terraform, representing a **limited automation identity** — like what you would use in CI/CD pipelines.

Why it matters:
Rather than using our full-access lab account, we want to simulate **realistic access boundaries** — mimicking how secure organizations prevent developers and automation tools from having unrestricted rights.

Environment variable usage:
Instead of hardcoding secrets, we inject them into the environment. This aligns with DevSecOps best practices and is supported natively by the Terraform AzureRM provider.

Steps

1. Navigate to **terraform.tfvars** and substitute <random unique suffix> with random letters and/or numbers of your choice to ensure name uniqueness.
2. Update line 2 of **main.tf**, replacing **<your subscription id>** with your lab subscription id.
3. Create the Service Principal scoped to your subscription (replace <sub_id> with your lab subscription:

   az ad sp create-for-rbac --name terraform-test-sp --role Reader --scopes /subscriptions/<sub_id>

4. Capture the following values from the output:

   appId → ARM_CLIENT_ID
   password → ARM_CLIENT_SECRET
   tenant → ARM_TENANT_ID
   subscriptionId → ARM_SUBSCRIPTION_ID

5. Set environment variables:

   $env:ARM_CLIENT_ID = "<appId>"
   $env:ARM_CLIENT_SECRET = "<password>"
   $env:ARM_SUBSCRIPTION_ID = "<sub_id>"
   $env:ARM_TENANT_ID = "<tenant_id>"

6. Adopt the SP identity:

   az logout

   az login --service-principal --username $env:ARM_CLIENT_ID --password=$env:ARM_CLIENT_SECRET --tenant $env:ARM_TENANT_ID

   Then run:

   az account show

   You should see "type": "servicePrincipal" in the output.

Phase 1 – Run Terraform With Limited SP Identity

Purpose

What we are doing:
Attempting to deploy infrastructure using Terraform under this limited SP identity.

Why it matters:
This phase is expected to fail — and that is intentional. It highlights that simply authenticating to Azure is not enough; **role assignments define what you are allowed to do**. It also encourages developers to review error messages meaningfully and connect failures to permissions.

Steps

1. Run:

   terraform init
   terraform plan
   terraform apply

2. Expect Terraform apply to **fail** with authorization errors.

Phase 2 – Assign Contributor Role to the Service Principal

Purpose

What we are doing:
Temporarily switching back to the lab account to assign Contributor rights to the SP.

Why it matters:
This demonstrates the **RBAC model in action** — where a principal's capabilities change dynamically based on role assignments. It mirrors how platform teams might approve temporary elevated access to unblock automation or development pipelines.

Steps

1. Re-adopt your lab identity:

   az logout
   az login

   Enter your student credentials in the launched browser session

2. Inserting your SP id and Subscription id, run:

   az role assignment create --assignee <appId> --role Contributor --scope /subscriptions/<sub_id>

3. Re-adopt the SP identity:

   az logout

   az login --service-principal --username $env:ARM_CLIENT_ID --password=$env:ARM_CLIENT_SECRET --tenant $env:ARM_TENANT_ID

4. Retry:

   terraform plan
   terraform apply

5. Terraform should now **succeed**.

Phase 3 – VM Size Precondition Check

Purpose

What we are doing:
Using Terraform's validation block inside a variable definition to restrict input values for VM sizes.

Why it matters:
In cloud environments with cost controls or policy restrictions, not all VM sizes are allowed. This validation protects the user from accidentally requesting expensive or disallowed SKUs — acting as a **client-side policy enforcement** before Terraform even contacts Azure.

Steps

1. Review the provided variables.tf file. Two variables are declared, vm_size and storage_account_name. Both have validations, the vm size must be one of the 4 listed and the storage account name must begin with "lab"

2. Review the provided terraform.tfvars file. Values for both variables are present. The vm size listed is one of the four permitted sizes and the storage account name begins with 'lab'.

3. In terraform.tfvars; change the vm size from **"Standard_B2s"**, to **"Standard_B4s"**

4. Run **terraform plan**. This should **fail** with an error indicating an invalid machine size choice…

```
Error: Invalid value for variable

  on terraform.tfvars line 1:
   1: vm_size = "Standard_B4S"

     var.vm_size is "Standard_B4S"

Only the following VM sizes are allowed in this lab: Standard_D2s_V3, Standard_DS2_V2, Standard_B2S, Standard_DS3_v2.

This was checked by the validation rule at variables.tf:4,3-13.
```

5. In terraform.tfvars; reset the vm size back to **Standard_B2s**

Phase 4 – Storage Account Name Validation

Purpose

What we are doing:

1. **Precondition** ensures the **input value** for the storage account name starts with "lab".
2. **Postcondition** ensures that after Terraform's logic runs, the **actual deployed name** still meets this rule.

Why it matters:
Sometimes, logic inside a Terraform configuration (e.g. using replace() or lower()) transforms a valid input into an invalid output.
The precondition validates the **user's input**, while the postcondition verifies the **final evaluated result**. This double-check ensures that even advanced logic does not break compliance rules unintentionally.

Steps

1. In terraform.tfvars; remove '**lab**' from the storage account name
2. Run **terraform plan.** This should fail pre-condition checks with an error indicating an invalid name...

```
Error: Invalid value for variable

  on terraform.tfvars line 2:
   2: storage_account_name = "storage3456123"

     var.storage_account_name is "storage3456123"

Storage account name must start with 'lab'.
```

3. Revert the storage account name so it begins with 'lab' and thus complies with the validation check. Planning should now succeed as before.
4. You will now introduce an error whereby the storage account name is altered so that, whilst the initial value passes validation check, the eventual value, altered by some logic processes, is not valid.

4. In main.tf, navigate to line 66 and change the **second** instance of ‘**lab**’ to ‘**test**’. This regenerates a new storage account name that varies from the name derived from the variable.

5. Run **terraform plan.** This should fail post-condition checks with an error indicating an invalid name…

```
Error: Resource postcondition failed

  on main.tf line 83, in resource "azurerm_storage_account" "storage":
  83:       condition     = can(regex("^lab", self.name))
    ├────────────────
    │ self.name is "teststorage3456123"

Evaluated storage name does not start with 'lab'.
```

6. Undo the changes to main.tf line 66

7. Run terraform plan followed by terraform apply to verify that the valid resources deploy successfully

Phase 5 – Cleanup

1. Run **terraform destroy**

2. Use **az logout** to log off as the Service Principal

3. Use **az logon** and log on using your lab account

4. Delete the Service Principal (optional): **az ad sp delete --id <appId>**

5. Clear environment variables:
   Remove-Item Env:ARM_CLIENT_ID
   Remove-Item Env:ARM_CLIENT_SECRET
   Remove-Item Env:ARM_TENANT_ID
   Remove-Item Env:ARM_SUBSCRIPTION_ID