Terragrunt — Using credentials dynamically and Azure multi-subscription support

Patrick Picard
Dec 21, 2020 · 4 min read

In this blog entry, I will combine a few topics as they are related:

- Multi-Account / Multi-Subscription support — Deploy parts of the environment to different subscriptions
- Using different credentials for parts of the infrastructure — Allows to follow a least privilege approach when deploying parts of the environment

This blog expands upon a previous entry discussing dependency management.

Multi-Account / Multi-Subscription support

When managing an enterprise scale cloud environment, it is important to divide it into functional pieces that will allow segregation of duties and minimizing blast radius. As such, core components such as platform management, networking, identity, audit, etc should be managed in separate subscriptions.

To deploy resources to different Azure subscriptions, it is possible to pass environment variables to Terraform and let it authenticate accordingly. In the code snippet below, there are 3 areas of interest:

- The dependency to the credential retrieval (discussed later in the article)
- The environment variables for the credentials (ARM_TENANT_ID, ARM_CLIENT_ID, ARM_CLIENT_SECRET)
- The subscription to pin the deployment. Set the value for ARM_SUBSCRIPTION_ID

The extra_arguments section requires special attention. Basically, this configuration says to inject the 4 environment variables when the following Terragrunt commands are called (init, apply,destroy, etc).

```
# Snippet Area #1 - Required to switch to appropriate identity
dependency "credentials" {
  config_path = "../credentials"
}

terraform {
  source = "git::ssh://git@github.com/xxxxxxx/terraform-azurerm-
core-iam.git"

extra_arguments "force_subscription" {
    commands = [
      "init",
      "apply",
      "destroy",
      "refresh",
      "import",
      "plan",
      "taint",
      "untaint"
    ]

# Snippet Area #2 - Passing environment variables to Terraform
env_vars = {
      ARM_TENANT_ID     = dependency.credentials.outputs.tenant_id
      ARM_CLIENT_ID     = dependency.credentials.outputs.client_id
      ARM_CLIENT_SECRET =
dependency.credentials.outputs.client_secret

# Snippet Area #3 - Passing the subscription ID to deploy resources

ARM_SUBSCRIPTION_ID = local.config.management.subscription_id
    }
  }
}
```

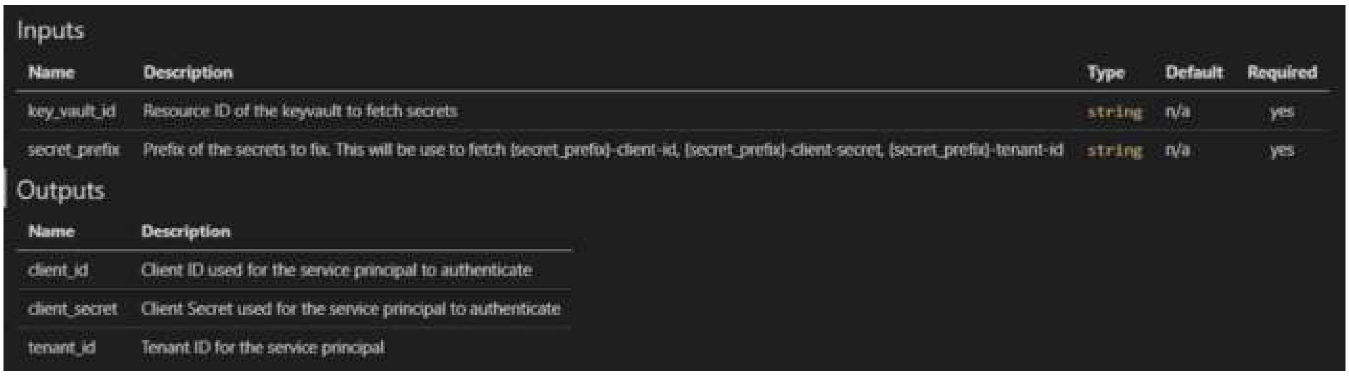
It is important to not miss any important actions in the commands array. I initially omitted “destroy” from the list which meant Terraform was using my logged in credentials and pinned subscription. So if I had my default subscription set to a different subscription than the terraform state, it would refresh the state and realize all the resources have been “deleted” and would remove them from the state…thus leaving a mess in the subscription that I had to clean manually. My battle scar is your fair warning!

Dynamic Credential Retrieval

As you deploy various components to multiple subscriptions, you can either use one service principal with full privileges or break it down to multiple service principals with more restrictive access. The latter is preferred to follow least privilege principles and reduce blast radius if the credential were to be compromised or an operator mistake.

As such, I had to design a way to retrieve credentials on the fly from Azure DevOps. Since I am working with Terraform and Terragrunt, I opted with a solution that felt Terraform native. I also wanted to avoid delegating this to the Azure DevOps with DevOps marketplace plugins just in case some of our clients were to use another CICD tool.

I built a credential fetcher that will retrieve 3 secrets from an Azure Key Vault to represent the tenant id, client id, and client secret. This Terraform module accepts 2 variables and outputs 3 values.



Inputs

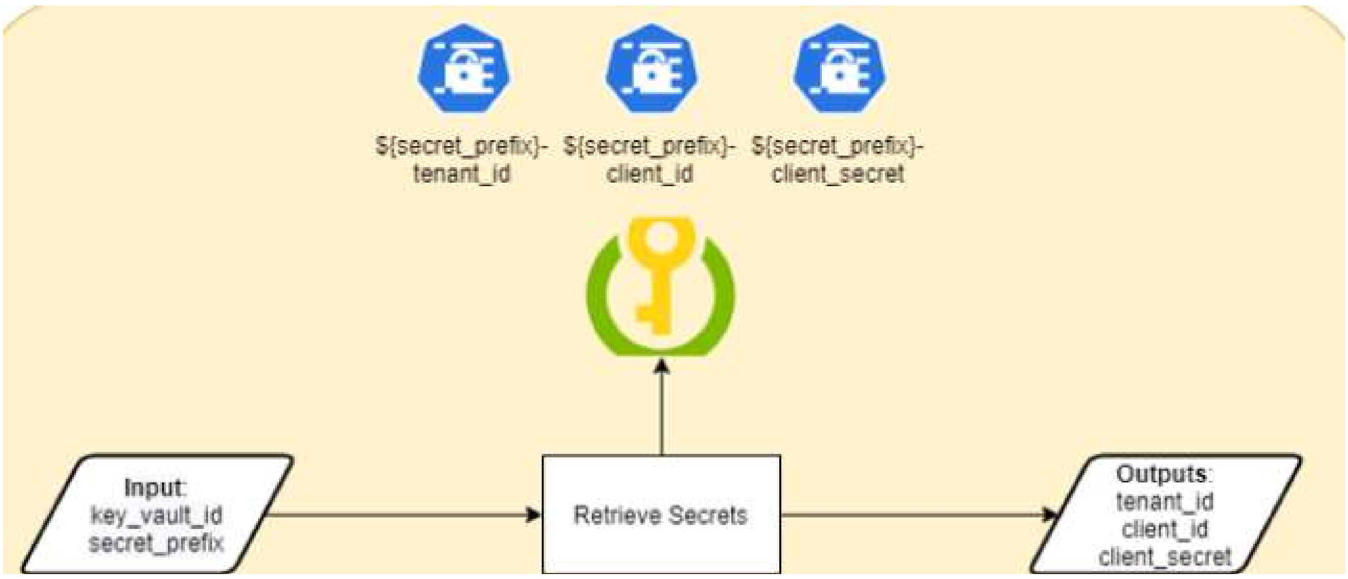
Name	Description	Type	Default	Required
key_vault_id	Resource ID of the keyvault to fetch secrets	string	n/a	yes
secret_prefix	Prefix of the secrets to fix. This will be use to fetch {secret_prefix}-client-id, {secret_prefix}-client-secret, {secret_prefix}-tenant-id	string	n/a	yes

Outputs

Name	Description
client_id	Client ID used for the service principal to authenticate
client_secret	Client Secret used for the service principal to authenticate
tenant_id	Tenant ID for the service principal

Credential Fetcher inputs and outputs

Here's a diagram of the Terraform module design. Probably one of the simplest Terraform module you can build!


${secret_prefix}-tenant_id
${secret_prefix}-client_id
${secret_prefix}-client_secret
Input: key_vault_id secret_prefix
Retrieve Secrets
Outputs: tenant_id client_id client_secret

To use the credential fetcher, a deployment must set a dependency to the credential fetcher. Then its outputs become available to pass as environment variables. If you refer to the code snippet above, see the *Snippet Area #2*.

Implications & Security

Terraform deployments use state files to capture the live state of a module which includes the inputs, details about all the objects under management, and values of the outputs of the module. Since the values from Azure Key Vault are sensitive, having them on the file system is not ideal thru a state file. As such, the filesystem should be encrypted, in transit encryption enabled, and very restrictive access from an RBAC perspective. Further, after each run, the local temporary files should be cleaned to prevent leaving sensitive content behind.

While I would like every deployments to only have the values in memory, it would add a lot of complexity to every module to fetch the credentials themselves.

We could have moved the credential retrieval to Azure DevOps and pass them in. But that would make the code specific to Azure DevOps and not easily able to retrieve different credentials for different components. We would end up with one pipeline per area of the total infrastructure…which I may revert to at some point and avoid the Terraform state issue.

Terragrunt Terraform Azure Azure Key Vault