WHITE PAPER: April 2025

Terraform and VMware® vDefend™ Firewall

Beginning Micro-segmentation as Code

Table of contents

Introduction

Terraform is a declarative tool for Infrastructure as Code (IaC). Resources are defined in manifests using HCL that represent objects in the infrastructure. These resources can then be managed by Terraform to execute lifecycle operations like create, read, update, and delete (CRUD). Terraform has providers written in Golang that are used to interact with product APIs like VMware NSX® Manager or VMware vCenter®. VMware® vDefend™ Distributed Firewall can be controlled using the Terraform provider to create groups, services, policy, rules and more. In this paper, we explore the basics of Terraform and the advantages of using it for IaC.

Infrastructure as Code vs Self-Service

Infrastructure as Code

Infrastructure as Code is used to provision objects and configuration of devices in the enterprise network infrastructure. They are commonly used by product owners and developers, much like code, to define what is required from the infrastructure for the application to run in the enterprise network. This can include configuration for routing and switching, load-balancing, security, and more. IaC, in most cases, uses a declarative model where all components are defined in a manifest, and a tool like Terraform will create all of the objects in the correct order it believes is required. Terraform manifests are idempotent, meaning, they can be run multiple times without fear of creating duplicate objects. This makes Terraform quite powerful and easier to use. This kind of behavior can be replicated using scripting languages, but it takes a lot more effort to ensure this result.

Self-Service

Self-Service operations are usually well defined, single run actions that make changes to a device in the enterprise network. State may or may not be tracked which can cause difficulty with lifecycle operations or duplicates. Self-Service operations are great for isolated day 2 actions that do not require the lifecycle upkeep or an IaC tool. Self-Service can be created using a scripting language like Python, Golang, Javascript, or an orchestration tool like VMware VCF Automation. Some popular self-service operations are Load-Balancer as a Service (LBaaS) or Firewall as a Service (FWaaS).

Terraform Basics

Provider

A provider interacts with the API of a device in the infrastructure. The NSX Manager has a provider called NSXT. This provider allows the devops engineer to perform create, update, read, and delete (CRUD) actions against the NSX Manager. This will be the provider used in these examples. The provider information can be placed in a single file with resources or in a separate file. For this paper, the provider information will be placed in a separate file called "`providers.tf`".

Files and Directories

All of the resources created in this paper will be placed in a file called "`main.tf`". Resources can be separated into multiple files and given any name, as long as they all reside in the same folder. The folder where all of the files live is called a root module. Child modules with specific functionality can be created and used by a root module. All Terraform projects have a root module, which is essentially the working directory where all the terraform manifest files live (.tf files).

Statefiles

Statefiles are files that keep track of the configuration of the devices the providers have configured using Terraform. The statefile records a mapping between the resources defined in the configuration file and the infrastructure actually provisioned in the product. It keeps getting updated with every terraform command. The device config is managed by terraform so edits or deletions of objects should not be performed manually in the device once they are created.

Terraform statefiles can hold sensitive information like passwords so they should **never** be checked into a git repository like Github or Gitlab. Statefiles can reside in the root module or remotely. If terraform manifests are separated into multiple folders, then statefiles will exist for each folder. These separate statefiles in different folders will not be able to share data between each other unless special "data" commands are used to import information. This paper will not cover how to do that.

Backend

Backends are where the statefiles are stored. By default, statefiles are stored locally in the same folder as the Terraform manifest files. Remote backends, like S3, Azure, Gitlab, and more can be used to store statefiles as well. Remote backends are typically used in more complex deployments and allow for Terraform modules to be run from many machines allowing them to maintain a consistent configuration state in the devices and objects Terraform managed. For the purposes of this paper, all backends will be local.

Authentication

In order to execute API calls against the NSX Manager, we need to use some type of credentials to authenticate to the NSX Manager. This is where the provider block comes into play.

```
provider "nsxt" {
    username = "admin"
    password = "password"
    allow_unverified_ssl = true
    host = "<YOUR_NSXMANAGER_FQDN_OR_IP_HERE>"
}
```

Putting your authentication credentials in your script is not recommended and should be avoided at all costs! The above portion was only included for completeness and should never be used as the preferred method in production. Terraform manifest files, or any other types of automation scripts, may eventually be checked into a repository. If sensitive data like usernames and passwords are defined in the file as plain text, anyone will be able to view those credentials. This can result in a serious security breach or allow unauthorized users the ability to make changes to the configuration using other means (like external API calls or the UI). Even if you overwrite and check in a newer version of the code, the repo will still have the history of the configuration and can still be collected by users. You will have to go through complicated processes to delete historical code commits which will result in lost history and potential confusion. The best policy, is to never hard-code sensitive data in any type of script(s) or application code in general.

With Terraform, more complicated methods like a Secrets Management Tool (Hashicorp Vault, AWS Secrets Manager, Google Secrets Manager, etc) or environment variables can be used for authentication. Secret Management Tools come with their own advantages and disadvantages as well and are well beyond the scope of this document. Please refer to a reputable Terraform book or resource for more information on best-practices. In this paper, environment variables will be utilized for authentication. This will avoid hard-coding sensitive data in the scripts and allow safe commits to code repositories without fear of exposing critical infrastructure credentials.

Below is a list of needed environment variables for the provider:

NSXT_ALLOW_UNVERIFIED_SSL	True
NSXT_MANAGER_HOST	<YOUR_NSXMANAGER_FQDN_OR_IP>
NSXT_USERNAME	<USERNAME_HERE>

NSXT_PASSWORD	<PASSWORD_HERE>

Please refer to your OS documentation to understand how to set environment variables.

Terraform documentation

Documentation for how to use Terraform is located at this link. The NSXT provider that will be used for this paper is located at this link. Only simple examples will be shown in this paper as this is meant to be a simple introduction.

Terraform with vDefend Firewall

A single directory will be created to contain the Terraform manifest files needed. The files are called "`providers.tf`" and "`main.tf`".

Providers.tf

```
terraform {
  required_providers {
    nsxt = {
      source = "vmware/nsxt"
    }
  }
}
```

"`required_providers`" tells Terraform what providers are absolutely needed in order for this to execute successfully. In more advanced situations, specific versions can be identified to prevent potential backwards compatibility issues as Terraform providers are improved upon. For this basic example, no version will be specified.

Main.tf

```
resource "nsxt_policy_group" "windows" {
  display_name = "Windows"
  description = "Group created by Terraform"
}
```

First, a resource is defined along with the type and ID. The type is "`nsxt_policy_group`" which is known as one of the resource types defined in the NSXT provider. Many aspects can be defined, but in this simple starting example, the "`display_name`" and "`description`" will be defined.

In order for the resource to be created, the backend and provider must be initialized.

```
terraform init

<--output-->
Initializing the backend...
Initializing provider plugins...
- Finding latest version of vmware/nsxt...
- Installing vmware/nsxt v3.8.0...
- Installed vmware/nsxt v3.8.0 (signed by a HashiCorp partner, key ID ED13BE650293896B)
Partner and community providers are signed by their developers.
If you'd like to know more about provider signing, you can read about it here:
https://www.terraform.io/docs/cli/plugins/signing.html
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
```

```
you run "terraform init" in the future.
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

The backend and provider has been successfully configured. Several folders and files are created. Most of these should NOT be checked into a git repo, especially the statefile. Now, to see what changes will be made without actually implementing them, run the "`plan`" command.

```
terraform plan
```

```
<--output-->
Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # nsxt_policy_group.windows will be created
  + resource "nsxt_policy_group" "windows" {
      + description  = "Group created by Terraform"
      + display_name = "Windows"
      + domain       = "default"
      + id           = (known after apply)
      + nsx_id       = (known after apply)
      + path         = (known after apply)
      + revision     = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these
actions if you run "terraform apply" now.
```

This is showing what will be created when the "`apply`" command is executed. It is important to understand that, to this point, nothing has been created in the NSX Manager by the provider. Terraform is showing what it will do when we decide to go forward with the plan. A plan file can be exported as well for review, but it is not required. Terraform can now be used to create the vDefend group in the NSX Manager

```
terraform apply
```

```
<--output-->
Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:
  # nsxt_policy_group.windows will be created
  + resource "nsxt_policy_group" "windows" {
      + description  = "Group created by Terraform"
      + display_name = "Windows"
      + domain       = "default"
      + id           = (known after apply)
      + nsx_id       = (known after apply)
      + path         = (known after apply)
      + revision     = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

nsxt_policy_group.windows: Creating...
nsxt_policy_group.windows: Creation complete after 1s [id=871962d3-a71a-4a6f-9b2d-b219391c730e]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

The vDefend group has now been created in the NSX Manager. It is managed completely by Terraform and will show in the statefile. The statefile should **NEVER** be changed manually. You can view the state of the object with some CLI commands. To see all managed objects, use:

```
terraform state list

<--output-->
nsxt_policy_group.windows
```

To view the state information of the object use:

```
terraform state show <object>
```

In this case it will be:

```
terraform state show nsxt_policy_group.windows

<--output-->
# nsxt_policy_group.windows:
resource "nsxt_policy_group" "windows" {
    description  = "Group created by Terraform"
    display_name = "Windows"
    domain       = "default"
    id           = "871962d3-a71a-4a6f-9b2d-b219391c730e"
    nsx_id       = "871962d3-a71a-4a6f-9b2d-b219391c730e"
    path         = "/infra/domains/default/groups/871962d3-a71a-4a6f-9b2d-b219391c730e"
    revision     = 0
}
```

To change the object, all that need to be done is edit the manifest. In this example, the "`display_name`" of the object will be changed:

```
resource "nsxt_policy_group" "windows" {
  display_name = "Windows TF"
  description = "Group created by Terraform"
}
```

Now run the "`plan`" command:

```
terraform plan

<--output-->
nsxt_policy_group.windows: Refreshing state... [id=871962d3-a71a-4a6f-9b2d-b219391c730e]

Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # nsxt_policy_group.windows will be updated in-place
  ~ resource "nsxt_policy_group" "windows" {
      ~ display_name = "Windows" -> "Windows TF"
        id           = "871962d3-a71a-4a6f-9b2d-b219391c730e"
        # (5 unchanged attributes hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.

terraform apply

<--output-->
nsxt_policy_group.windows: Refreshing state... [id=871962d3-a71a-4a6f-9b2d-b219391c730e]

Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # nsxt_policy_group.windows will be updated in-place
  ~ resource "nsxt_policy_group" "windows" {
      ~ display_name = "Windows" -> "Windows TF"
        id           = "871962d3-a71a-4a6f-9b2d-b219391c730e"
        # (5 unchanged attributes hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

nsxt_policy_group.windows: Modifying... [id=871962d3-a71a-4a6f-9b2d-b219391c730e]
nsxt_policy_group.windows: Modifications complete after 0s [id=871962d3-a71a-4a6f-9b2d-b219391c730e]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

The vDefend group's "`display_name`" has now been changed.

To destroy the resource, use the following command:

```
terraform destroy

<--output-->
nsxt_policy_group.windows: Refreshing state... [id=871962d3-a71a-4a6f-9b2d-b219391c730e]

Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # nsxt_policy_group.windows will be destroyed
  - resource "nsxt_policy_group" "windows" {
      - description  = "Group created by Terraform" -> null
      - display_name = "Windows TF" -> null
      - domain       = "default" -> null
      - id           = "871962d3-a71a-4a6f-9b2d-b219391c730e" -> null
      - nsx_id       = "871962d3-a71a-4a6f-9b2d-b219391c730e" -> null
      - path         = "/infra/domains/default/groups/871962d3-a71a-4a6f-9b2d-b219391c730e" -> null
      - revision     = 1 -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

nsxt_policy_group.windows: Destroying... [id=871962d3-a71a-4a6f-9b2d-b219391c730e]
nsxt_policy_group.windows: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

The resource is now gone and nothing will be returned when this follow command is executed:

```
terraform state list
```

Policy with a Rule

In the last example, two groups and a policy with a single rule will be created. All of the information will be placed in the same "`main.tf`" file.

Main.tf

```
resource "nsxt_policy_group" "windows" {
  display_name = "Windows"
  description = "Group created by Terraform"
}

resource "nsxt_policy_group" "active_directory" {
  display_name = "AD Servers"
  description = "All AD Servers"
}

resource "nsxt_policy_security_policy" "adpolicy" {
  display_name = "AD Policy"
  description = "Rules for AD Servers"
  category = "Application"

  rule {
    display_name = "Windows AD Rule"
    action = "ALLOW"
    source_groups = [
        nsxt_policy_group.windows.path
    ]
    destination_groups = [
        nsxt_policy_group.active_directory.path
    ]
    services = [
        "/infra/services/SSH"
    ]
    logged = true
    scope = [
        nsxt_policy_group.windows.path,
        nsxt_policy_group.active_directory.path
     ]
  }
}
```

In the example above, two groups are created. Those groups are then utilized in the source, destination, and scope (for the Applied-To) field in the subsequent rule defined in the new policy. Notice how the groups are referenced by their type "`nsxt_policy_group`" and managed resource id "`windows`" and "`active_directory`". The path is required for the source, destination, and scope so the "`path`" value is referenced from each group that will be created. The path will be known once the groups are created but Terraform only needs to know the reference and it will handle the details once the groups are created. This is the same for every other property that shows "`known after apply`" below. Terraform knows the groups will need to be created first before the policy, so it will handle creation order as well. Now the "`plan`" and "`apply`" commands can be run.

```
terraform plan

<--output-->
Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # nsxt_policy_group.active_directory will be created
  + resource "nsxt_policy_group" "active_directory" {
      + description  = "All AD Servers"
      + display_name = "AD Servers"
      + domain       = "default"
      + id           = (known after apply)
      + nsx_id       = (known after apply)
      + path         = (known after apply)
      + revision     = (known after apply)
    }

  # nsxt_policy_group.windows will be created
  + resource "nsxt_policy_group" "windows" {
      + description  = "Group created by Terraform"
      + display_name = "Windows"
      + domain       = "default"
      + id           = (known after apply)
      + nsx_id       = (known after apply)
      + path         = (known after apply)
      + revision     = (known after apply)
    }
  # nsxt_policy_security_policy.adpolicy will be created
  + resource "nsxt_policy_security_policy" "adpolicy" {
      + category        = "Application"
      + description     = "Rules for AD Servers"
      + display_name    = "AD Policy"
      + domain          = "default"
      + id              = (known after apply)
      + locked          = false
      + nsx_id          = (known after apply)
      + path            = (known after apply)
      + revision        = (known after apply)
      + sequence_number = 0
      + stateful        = true
      + tcp_strict      = (known after apply)

      + rule {
          + action                = "ALLOW"
          + destination_groups    = (known after apply)
          + destinations_excluded = false
          + direction             = "IN_OUT"
          + disabled              = false
          + display_name          = "Windows AD Rule"
          + ip_version            = "IPV4_IPV6"
          + logged                = true
          + nsx_id                = (known after apply)
          + path                  = (known after apply)
          + revision              = (known after apply)
          + rule_id               = (known after apply)
          + scope                 = (known after apply)
          + sequence_number       = (known after apply)
          + services              = [
              + "/infra/services/SSH",
            ]
          + source_groups         = (known after apply)
          + sources_excluded      = false
        }
    }

Plan: 3 to add, 0 to change, 0 to destroy.

terraform apply

<--output-->
Terraform used the selected providers to generate the following execution plan. Resource actions are
indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # nsxt_policy_group.active_directory will be created
  + resource "nsxt_policy_group" "active_directory" {
      + description  = "All AD Servers"
      + display_name = "AD Servers"
      + domain       = "default"
      + id           = (known after apply)
      + nsx_id       = (known after apply)
      + path         = (known after apply)
      + revision     = (known after apply)
    }

  # nsxt_policy_group.windows will be created
  + resource "nsxt_policy_group" "windows" {
      + description  = "Group created by Terraform"
      + display_name = "Windows"
      + domain       = "default"
      + id           = (known after apply)
      + nsx_id       = (known after apply)
      + path         = (known after apply)
      + revision     = (known after apply)
    }

  # nsxt_policy_security_policy.adpolicy will be created
  + resource "nsxt_policy_security_policy" "adpolicy" {
      + category        = "Application"
      + description     = "Rules for AD Servers"
      + display_name    = "AD Policy"
      + domain          = "default"
      + id              = (known after apply)
      + locked          = false
      + nsx_id          = (known after apply)
      + path            = (known after apply)
      + revision        = (known after apply)
      + sequence_number = 0
      + stateful        = true
      + tcp_strict      = (known after apply)

      + rule {
          + action                = "ALLOW"
          + destination_groups    = (known after apply)
          + destinations_excluded = false
          + direction             = "IN_OUT"
          + disabled              = false
          + display_name          = "Windows AD Rule"
          + ip_version            = "IPV4_IPV6"
          + logged                = true
          + nsx_id                = (known after apply)
          + path                  = (known after apply)
          + revision              = (known after apply)
          + rule_id               = (known after apply)
          + scope                 = (known after apply)
          + sequence_number       = (known after apply)
          + services              = [
              + "/infra/services/SSH",
            ]
          + source_groups         = (known after apply)
          + sources_excluded      = false
        }
    }
```

```
Plan: 3 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
nsxt_policy_group.active_directory: Creating...
nsxt_policy_group.windows: Creating...
nsxt_policy_group.windows: Creation complete after 0s [id=c9a0c4e1-4611-4ab6-8f0a-d7e0e40454b5]
nsxt_policy_group.active_directory: Creation complete after 0s [id=0af10c63-8675-4ff2-8715-ae31bb1bee8d]
nsxt_policy_security_policy.adpolicy: Creating...
nsxt_policy_security_policy.adpolicy: Creation complete after 1s [id=476f42a8-2058-48b1-a748-323dabae916e]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

Summary

VMware vDefend Firewall objects can easily be managed using Terraform for Infrastructure as Code (IaC) deployments. These strategies can be enhanced to deploy complex network and security deployments in the NSX Manager using scripts, CI/CD pipelines, VMware VCF Automation, and more. It is up to the DevOps engineer to develop the automation strategy that will meet the requirements of the enterprise.