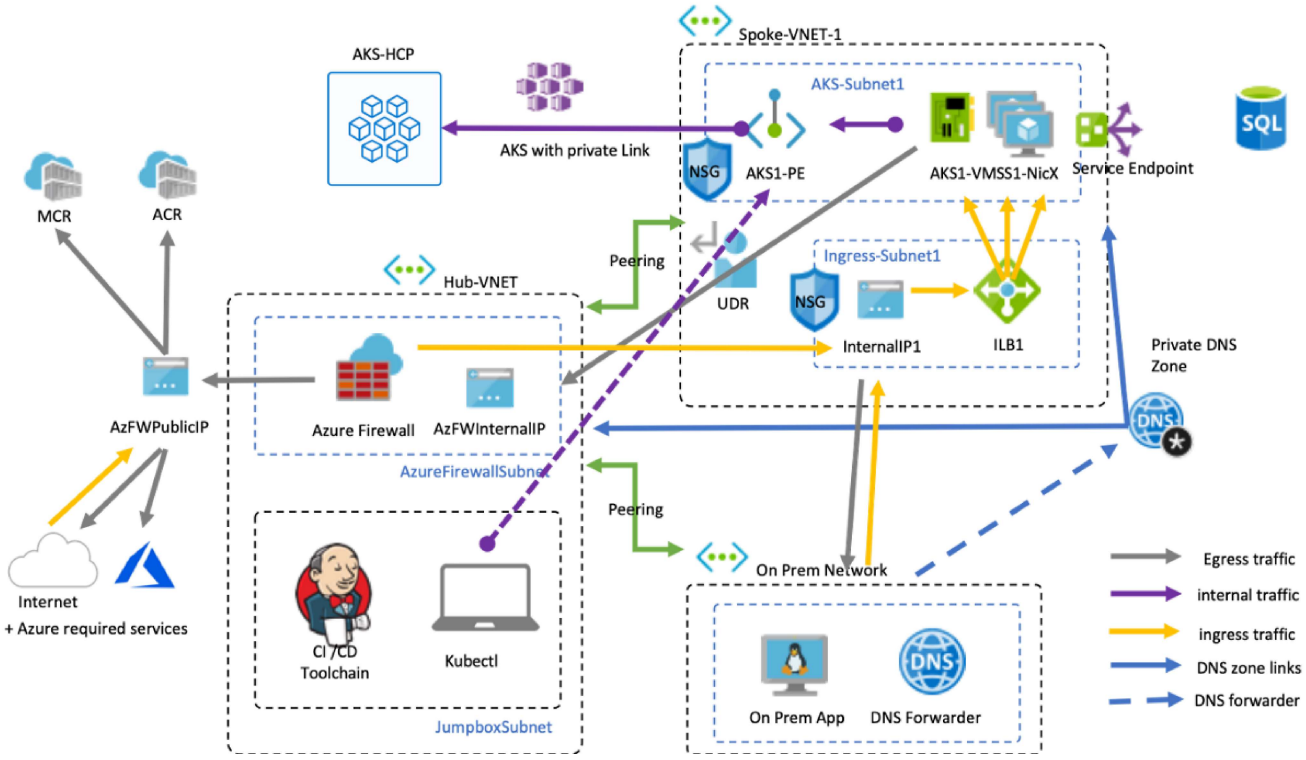
Create a “Fully private” AKS infrastructure with Terraform

Pavel Tuzov
Jun 24, 2020 · 7 min read ★



Infrastructure from https://medium.com/@denniszielke/fully-private-aks-clusters-without-any-public-ips-finally-7f5688411184

In Azure you can create a **private** AKS cluster, in which the traffic between the node pools and the API server does not leave the private network. But often, making only the communication between nodes and the control plane private is not enough for your security needs. In a *fully* private cluster you also want to not expose and use public IPs.

Such an architecture is described in details in this article:

> Fully private AKS clusters — without any public ips — finally!

One year after my last post on leveraging an azure firewall I want to

medium.com

Here I will describe the Terraform script which will manage the exact infrastructure as in the above post. I will explain only occasionally why we need the different Azure resources, so you probably want to read Dennis’s article first.

I find this project also a very good example to practice different Terraform features:

- organizing the project into modules
- defining terraform and provider versions
- type constraints
- each and for_each expressions
- functions
- defining explicit dependency
- and even using provisioners!

The sources can be found in this repository: https://github.com/patuzov/terraform-private-aks

Resource groups

There will be two separate resource groups for the hub and spoke networks. So, here they are in the *main.tf,* along with the *terraform* and *provider* blocks.

Note, we will need azurerm provider starting with version 2.5, as this is when the ***outbound_type*** *argument was introduced for AKS cluster resource.*

```
terraform {
  required_version = ">= 0.12"
}

provider "azurerm" {
  version = "~>2.5"
```

```
  features {}
}

resource "azurerm_resource_group" "vnet" {
  name     = var.vnet_resource_group_name
  location = var.location
}

resource "azurerm_resource_group" "kube" {
  name     = var.kube_resource_group_name
  location = var.location
}
```

```
terraform init
terraform apply
```

…and we can see our two resource groups in the Azure Portal.

Networking

VNETs, Subnets

As for the networks, we will need a VNET and one or more SUBNETs for both: hub and spoke. This is a great opportunity to use **Terraform Modules**. Using modules, you can hide complexity and, most importantly, reuse code in Terraform.

We will create a *modules* folder and a *vnet* folder inside it, with all the usual files for a module: *main.tf, variables.tf, outputs.tf.*

There is nothing fancy for the virtual network resource. Just set the values provided via the input variables:

```
resource "azurerm_virtual_network" "vnet" {
  name                = var.vnet_name
  address_space       = var.address_space
  location            = var.location
  resource_group_name = var.resource_group_name
}
```

main.tf hosted with ♡ by GitHub view raw

Now we might want to create a list of subnets in this vnet and we need to receive following input values for each subnet: name and the IP address range. We could receive as input a list of objects containing this information:

```
variable subnets {
  description = "Subnets configuration"
  type = list(object({
    name             = string
    address_prefixes = list(string)
  }))
}
```

variables.tf hosted with ♡ by GitHub view raw

…which, in my opinion, makes sense for the *interface* of this module.

For each such object a subnet resource can be created, using the *for_each* expression. But this expression expects either a list of strings or a map. So we’ll have to transform our list of objects to a map. *for* expression comes to the rescue:

```
resource "azurerm_subnet" "subnet" {
  for_each = { for subnet in var.subnets : subnet.name => subnet.address_prefixes }

  name                 = each.key
  resource_group_name  = var.resource_group_name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = each.value
}
```

main.tf hosted with ♡ by GitHub view raw

The *for* expression will create a map with *subnet.name* as keys and *subnet.address_prefixes* as corresponding values. For each such key/value pair, an *azurerm_subnet* resource will be created.

From this module, we will need vnet and subnet IDs. We define this in the *outputs.tf*. The same *technique* with the *for* expression is used to get back the subnet IDs as a map:

```
output vnet_id {
  description = "Generated VNET ID"
  value       = azurerm_virtual_network.vnet.id
}

output subnet_ids {
```

```
  description = "Generated subnet IDs map"
  value       = { for subnet in azurerm_subnet.subnet : subnet.name => subnet.id }
}
```

outputs.tf hosted with ♡ by GitHub view raw

Finally, we make use of this module in our *main.tf*:

```
module "hub_network" {
  source              = "./modules/vnet"
  resource_group_name = azurerm_resource_group.vnet.name
  location            = var.location
  vnet_name           = var.hub_vnet_name
  address_space       = ["10.0.0.0/22"]
  subnets = [
    {
      name : "AzureFirewallSubnet"
      address_prefixes : ["10.0.0.0/24"]
    },
    {
      name : "jumpbox-subnet"
      address_prefixes : ["10.0.1.0/24"]
    }
  ]
}

module "kube_network" {
  source              = "./modules/vnet"
  resource_group_name = azurerm_resource_group.kube.name
  location            = var.location
  vnet_name           = var.kube_vnet_name
  address_space       = ["10.0.4.0/22"]
  subnets = [
    {
      name : "aks-subnet"
      address_prefixes : ["10.0.5.0/24"]
    }
  ]
}
```

You need to run `terraform init` again to initialize the new module. After that you can *apply* and the VNETs are in the portal as well, under the corresponding resource groups.

Peering

Now we need to peer the VNETs. The *azurerm* provider has the *azurerm_virtual_network_peering* resource. But, as the peering will have to be done both ways, it’s better to abstract this in a new module.

Here is the very simple module’s *main.tf* code:

```
resource "azurerm_virtual_network_peering" "peering" {
  name                      = var.peering_name_1_to_2
  resource_group_name       = var.resource_group
  virtual_network_name      = var.vnet_1_name
  remote_virtual_network_id = var.vnet_2_id
}

resource "azurerm_virtual_network_peering" "peering-back" {
  name                      = var.peering_name_2_to_1
  resource_group_name       = var.resource_group
  virtual_network_name      = var.vnet_2_name
  remote_virtual_network_id = var.vnet_1_id
}
```

main.tf hosted with ♡ by GitHub view raw

And this is how we use it:

```
module "vnet_peering" {
  source              = "./modules/vnet_peering"
  vnet_1_name         = var.hub_vnet_name
  vnet_1_id           = module.hub_network.vnet_id
  vnet_1_rg           = azurerm_resource_group.vnet.name
  vnet_2_name         = var.kube_vnet_name
  vnet_2_id           = module.kube_network.vnet_id
  vnet_2_rg           = azurerm_resource_group.kube.name
  peering_name_1_to_2 = "HubToSpoke1"
  peering_name_2_to_1 = "Spoke1ToHub"
}
```

main.tf hosted with ♡ by GitHub view raw

Note how we use outputs of one module as inputs for the other.

Don’t forget to `terraform init`, in order to initialize the new module.

If you apply this configuration, you will see the *Peerings* settings under both VNETs.

Firewall

An Azure Firewall will need a public IP and we’ll also configure the exceptions for all the *external access* AKS needs in order to function properly (like pulling images or OS updates). So why not hide it all into yet another module?

Here is the code for the public IP, firewall itself and an example of a network rule for the *aks basics* (there are more rule collections defined — see the git repo):

```
resource "azurerm_public_ip" "pip" {
  name                = var.pip_name
  resource_group_name = var.resource_group
  location            = var.location
  allocation_method   = "Static"
  sku                 = "Standard"
}

resource "azurerm_firewall" "fw" {
  name                = var.fw_name
  location            = var.location
  resource_group_name = var.resource_group

  ip_configuration {
    name                 = "fw_ip_config"
    subnet_id            = var.subnet_id
    public_ip_address_id = azurerm_public_ip.pip.id
  }
}

resource "azurerm_firewall_application_rule_collection" "aksbasics" {
  name                = "aksbasics"
  azure_firewall_name = azurerm_firewall.fw.name
  resource_group_name = var.resource_group
  priority            = 101
  action              = "Allow"

  rule {
    name             = "allow network"
    source_addresses = ["*"]

    target_fqdns = [
      "*.cdn.mscr.io",
      "mcr.microsoft.com",
      "*.data.mcr.microsoft.com",
      "management.azure.com",
      "login.microsoftonline.com",
      "acs-mirror.azureedge.net",
      "dc.services.visualstudio.com",
```

```
        "*.opinsights.azure.com",
        "*.oms.opinsights.azure.com",
        "*.microsoftonline.com",
        "*.monitoring.azure.com",
      ]

      protocol {
        port = "80"
        type = "Http"
      }

      protocol {
        port = "443"
        type = "Https"
      }
    }
```

```
module "firewall" {
  source         = "./modules/firewall"
  resource_group = azurerm_resource_group.vnet.name
  location       = var.location
  pip_name       = "azureFirewalls-ip"
  fw_name        = "kubenetfw"
  subnet_id      = module.hub_network.subnet_ids["AzureFirewallSubnet"]
}
```

main.tf hosted with ♡ by GitHub view raw

Note how we get the Azure Firewall Subnet ID from the network module’s output.

Route Table

Finally, we’ll create a user defined route to force all traffic (0.0.0.0/0) from the AKS subnet to the firewall (specifically, to its private IP). For this, we need to create the route table with the corresponding route and associate it to the AKS subnet. And, yes, we’ll create a new module for this:

```
resource "azurerm_route_table" "rt" {
  name                = var.rt_name
  location            = var.location
  resource_group_name = var.resource_group

  route {
    name                   = "kubenetfw_fw_r"
```

```
    address_prefix         = "0.0.0.0/0"
    next_hop_type          = "VirtualAppliance"
    next_hop_in_ip_address = var.firewal_private_ip
  }
}

resource "azurerm_subnet_route_table_association" "aks_subnet_association" {
  subnet_id      = var.subnet_id
  route_table_id = azurerm_route_table.rt.id
}
```

And use the module:

```
module "routetable" {
  source             = "./modules/route_table"
  resource_group     = azurerm_resource_group.vnet.name
  location           = var.location
  rt_name            = "kubenetfw_fw_rt"
  r_name             = "kubenetfw_fw_r"
  firewal_private_ip = module.firewall.fw_private_ip
  subnet_id          = module.kube_network.subnet_ids["aks-subnet"]
}
```

main.tf hosted with ♡ by GitHub view raw

AKS

We are finally ready to define the AKS cluster!

```
resource "azurerm_kubernetes_cluster" "privateaks" {
  name                    = "private-aks"
  location                = var.location
  kubernetes_version      = "1.16.9"
  resource_group_name     = var.kube_resource_group_name
  dns_prefix              = "private-aks"
  private_cluster_enabled = true

  default_node_pool {
    name           = "default"
    node_count     = var.nodepool_nodes_count
    vm_size        = var.nodepool_vm_size
    vnet_subnet_id = module.kube_network.subnet_ids["aks-subnet"]
  }

  identity {
    type = "SystemAssigned"
```

```
  }

  network_profile {
    docker_bridge_cidr = var.network_docker_bridge_cidr
    dns_service_ip     = var.network_dns_service_ip
    network_plugin     = "azure"
    outbound_type      = "userDefinedRouting"
    service_cidr       = var.network_service_cidr
  }

  depends_on = [module.routtable]
}

# https://github.com/Azure/AKS/issues/1557
resource "azurerm_role_assignment" "vmcontributor" {
  role_definition_name = "Virtual Machine Contributor"
  scope                = azurerm_resource_group.vnet.id
  principal_id         = azurerm_kubernetes_cluster.privateaks.identity[0].principal_id
}
```

There are 2 important things to mention here:

- Enabling the actual private AKS cluster feature: `private_cluster_enabled = true`. This will also create a *Private DNS Zone* resource, which will have an *A Record* pointing to the **private** IP address of the API, allowing *everyone* from the AKS VNET (Spoke) to resolve the control plane’s address.
- Setting *outbound type* to *userDefinedRouting*, which will prevent creating a public IP for the cluster’s egress traffic and make use of the “user defined” Route Table: `outbound_type = "userDefinedRouting"` .

Also, did you notice the `depends_on` argument? Terraform can identify if a resource depends on another. For example, in order to deploy this AKS cluster in the “aks-subnet” subnet, Terraform knows it has to create the vnet and subnet first. These are the *implicit* dependencies. If Terraform doesn’t *see* dependencies between resources, it will [luckily] create them in parallel.

But in our case, as we set the *userDefinedRouting* outbound type, AKS will expect to have a route table associated with its subnet, so it actually depends on the route table resource. We can let Terraform know about this dependency by *explicitly* defining it with the `depends_on` argument.

Also, because of the still unresolved issue, we need to allow the created Service Prcinipal (by the managed identity) to edit the user defined routes in the route table from the *vnet* resource group, by defining the *azure role assignment*.

And we’re done! We now have an AKS cluster, in which the API doesn’t have a public endpoint and communicates with the nodes only through the Azure Backbone. The cluster also doesn’t expose and doesn’t use any public IPs (except those that we allowed it to).

Jumpbox

The problem is, since the control plane doesn’t expose any public endpoint, you will not be able to manage your cluster, say, with kubectl from your local machine, or even from the Azure DevOps pipelines (if you’re using Microsoft-hosted agents).

Remember the *Private DNS Zone* from the previous step? Only the Spoke VNET can resolve the API. But we can link other VNETs too. So, let’s create a (Linux) Virtual Machine in the Hub VNET, install kubectl on it and link the VNET in the Private DNS Zone.

No surprises here: I’ll *hide* it all in a separate module.

I’ll skip all the declarations for public IP, network security group, etc. You can find them in the github repo under *jumpbox* module. Here are the most important things:

```
resource "azurerm_linux_virtual_machine" "jumpbox" {
  name                            = "jumpboxvm"
  location                        = var.location
  resource_group_name             = var.resource_group
  network_interface_ids           = [azurerm_network_interface.vm_nic.id]
  size                            = "Standard_DS1_v2"
  computer_name                   = "jumpboxvm"
  admin_username                  = var.vm_user
  admin_password                  = random_string.adminpassword.result
  disable_password_authentication = false

  os_disk {
    name                 = "jumpboxOsDisk"
    caching              = "ReadWrite"
    storage_account_type = "Premium_LRS"
  }

  source_image_reference {
    publisher = "Canonical"
```

```
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04.0-LTS"
    version   = "latest"
  }

  provisioner "remote-exec" {
    connection {
      host     = self.public_ip_address
      type     = "ssh"
      user     = var.vm_user
      password = random_string.adminpassword.result
    }

    inline = [
      "sudo apt-get update && sudo apt-get install -y apt-transport-https gnupg2",
      "curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -",
      "echo 'deb https://apt.kubernetes.io/ kubernetes-xenial main' | sudo tee -a /etc/apt/sour
      "sudo apt-get update",
      "sudo apt-get install -y kubectl",
      "curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash"
    ]
  }
}

resource "azurerm_private_dns_zone_virtual_network_link" "hublink" {
  name                  = "hubnetdnsconfig"
  resource_group_name   = var.dns_zone_resource_group
  private_dns_zone_name = var.dns_zone_name
  virtual_network_id    = var.vnet_id
}
```

random_string resource) and show it in the terraform output, so that I can later ssh to the VM.

Probably you noticed a block in the VM resource which is a *taboo* in Terraform: **remote-exec provisioner**. Terraform provides them and kindly asks you not to use them. But, in this case, it fits perfectly! We’re spinning up this VM just to test that we can connect to the control plane with kubectl from our private network. And, being lazy to do it manually, I’ll let the provisioner automatically install kubectl and azure cli when it creates the VM.

As mentioned previously, we also need to link the hub VNET to the Private DNS Zone (which is automatically created with the AKS cluster). We can easily achieve this with

the *azurerm_private_dns_zone_virtual_network_link* resource.

From this module, we will need the following information:

```
output "jumpbox_ip" {
  description = "Jumpbox VM IP"
  value       = azurerm_linux_virtual_machine.jumpbox.public_ip_address
}

output "jumpbox_username" {
  description = "Jumpbox VM username"
  value       = var.vm_user
}

output "jumpbox_password" {
  description = "Jumpbox VM admin password"
  value       = random_string.adminpassword.result
}
```

outputs.tf hosted with ♡ by GitHub view raw

To use this module, we have a challenge: getting the Private DNS Zone’s name. It is generated automatically by Azure when the private AKS cluster is created (it’s not directly maintained by Terraform) and it even contains a GUID in it:

9c3df69f-b281–42ef-9996–89941b484906.privatelink.westeurope.azmk8s.io

Fortunately, it’s not that random. If you look at the private FQDN of the API, you will see this:

*private-aks-7b099028.**9c3df69f-b281–42ef-9996–89941b484906.privatelink.westeurope.azmk8s.io***

And this information is known by Terraform. We just need to extract the private DNS Zone’s name from it, using various Terraform functions: *join, slice, split* and *length:*

```
join(".", slice(split(".",
azurerm_kubernetes_cluster.privateaks.private_fqdn), 1,
length(split(".",
azurerm_kubernetes_cluster.privateaks.private_fqdn))))
```

This is how the module is used:

```
module "jumpbox" {
  source                  = "./modules/jumpbox"
  location                = var.location
  resource_group          = azurerm_resource_group.vnet.name
  vnet_id                 = module.hub_network.vnet_id
  subnet_id               = module.hub_network.subnet_ids["jumpbox-subnet"]
  dns_zone_name           = join(".", slice(split(".", azurerm_kubernetes_cluster.privateaks.pri
  dns_zone_resource_group = azurerm_kubernetes_cluster.privateaks.node_resource_group
}
```

main.tf hosted with ♡ by GitHub view raw

Lastly, we want to define our outputs:

```
output "ssh_command" {
  value = "ssh ${module.jumpbox.jumpbox_username}@${module.jumpbox.jumpbox_ip}"
}

output "jumpbox_password" {
  description = "Jumpbox Admin Passowrd"
  value       = module.jumpbox.jumpbox_password
}
```

outputs.tf hosted with ♡ by GitHub view raw

After applying the entire configuration, we get an output like this:

```
Outputs:

jumpbox_password = (nSq7qMxbJ
ssh_command = ssh azureuser@52.157.213.177
```

Terraform output

Now we can ssh into the Linux VM with kubectl and azure cli already installed. All what’s left is authenticating with Azure, getting the configuration for kubectl and we can connect to the API:

```
azureuser@jumpboxvm:~$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-default-75302467-vmss000000   Ready    agent   11h   v1.16.9
```

Using kubectl with a private AKS cluster

The traffic from the pods will be redirected to Azure Firewall and corresponding rules will be applied. Let’s check it.

In the *firewall* module, a *test* rule is defined, which will allow traffic to *bing.com*:

```
resource "azurerm_firewall_application_rule_collection" "test" {
  name                = "test"
  azure_firewall_name = azurerm_firewall.fw.name
  resource_group_name = var.resource_group
  priority            = 104
  action              = "Allow"

  rule {
    name             = "allow network"
    source_addresses = ["*"]

    target_fqdns = [
      "*.bing.com"
    ]

    protocol {
      port = "80"
      type = "Http"
    }

    protocol {
      port = "443"
      type = "Https"
    }
  }
}
```

Let’s create a pod and try to make a couple of requests from its container:

```
azureuser@jumpboxvm:~$ kubectl run busy --image=busybox -- sleep "3600"
pod/busy created
azureuser@jumpboxvm:~$ kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
busy   1/1     Running   0          9s
azureuser@jumpboxvm:~$ kubectl exec -it busy -- sh
/ # wget http://www.google.com
Connecting to www.google.com (172.217.17.68:80)
wget: server returned error: HTTP/1.1 470 status code 470
/ #
```

```
/ #
/ # wget http://www.bing.com
Connecting to www.bing.com (204.79.197.200:80)
saving to 'index.html'
index.html           100% |********************************************
'index.html' saved
/ #
```

Making external http requests from a container

Exactly what we wanted to achieve: external traffic is blocked, unless we have explicit rules to allow it.

Terraform Kubernetes Azure Kubernetes Service Azure