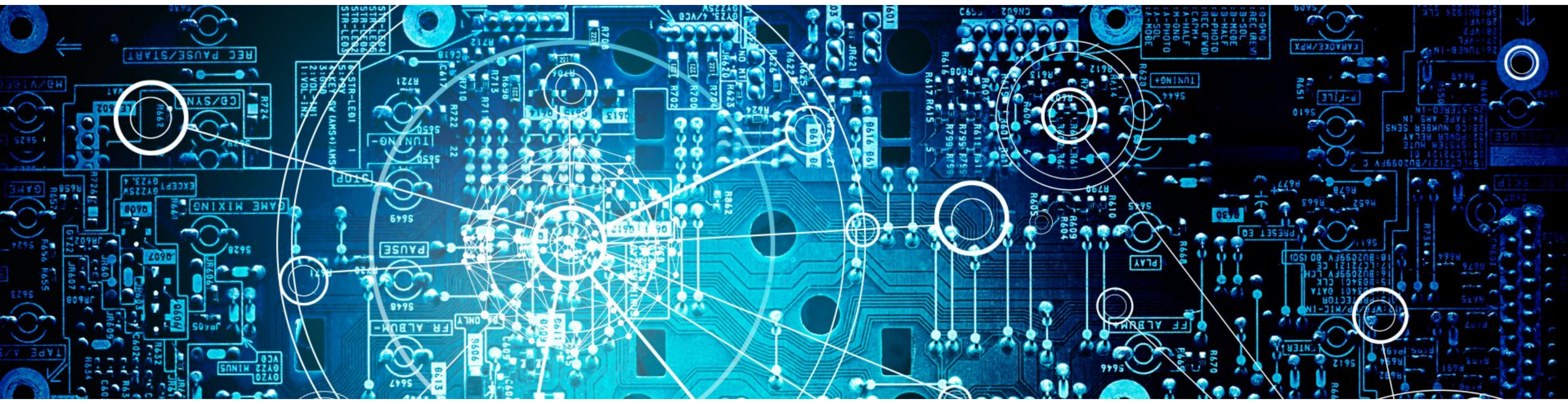


AZURE COMPLIANCE-AS-CODE UNIT-TESTING WITH GOLANG

June 30, 2022 Romain Aviolat cloud-native-security

Leave a comment

Infrastructure-as-Code (IaC) is great. It allows teams to deploy infrastructure quickly in a consistent and repeatable manner and when coupled with a proper CI/CD process (linting, SAST (Static Application Security

Testing), 4-eyes reviews, multi-env, …) it creates a powerful security framework for platform and development teams.

However as great as IaC is, it does not prevent users from making mistakes; it is easy to wrongly expose a service and thus increase your attack surface or to deploy it in a way that conflicts with your compliance requirements.

Luckily Cloud providers offer guardrail functionalities to catch those mistakes and to enforce your compliance requirements.

Azure provides it under the Azure Policy feature; for Google it is called Organization Policies and for AWS (Amazon Web Services) it is Service Control Policies. The implementation methods and features differ between the different cloud providers, but the idea is the same; provide central control over the maximum available permissions for the accounts in your organization.

In this blog post I'll cover how you can manage these guardrails as-code and how you can code unit-tests to validate them. We will focus on the Azure platform, but it could be easily reproduced on the others.

(all the code present is this blog-post is available on the following Github repository)

Scoping guardrails

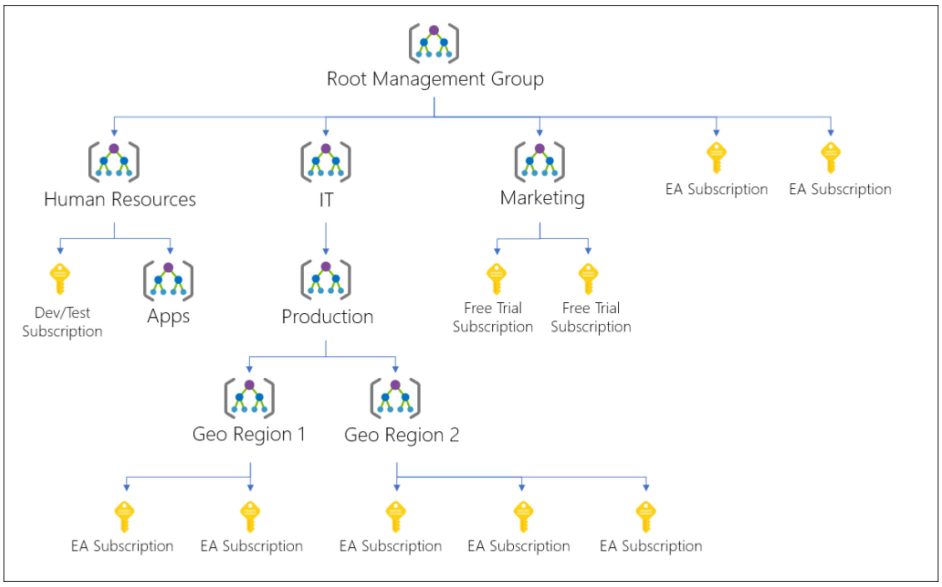
You need to start somewhere, and the blank page syndrome can be hard to overcome… My recommendation is to start with low hanging fruits. First, things that you never want to see happening in your production environment no matter what.

Ideally you should also pick things that will have little to no impact on your platform and development teams – it’s always frustrating to rollback or make people angry at the beginning of a new project. Also, keep in mind that you will need the support of these teams to progress on your Cloud-native security journey, so it is best if you have their buy-in day 1.

We’ll start with two simple use-cases:

1. We’ll forbid the deployment of resources in certain regions.
2. We’ll forbid certain source-addresses / destination ports patterns when creating Network Security Groups (NSG).

Now we need to define where in the accounts hierarchy we want to apply these policies. Azure recommends organizing Subscriptions under Management Groups. In this blog post that’s where we’ll assign them, but they could be just as readily assigned at the Subscriptions level or at the Resource Groups level as well. Assigning policies at the top of the tree means that all the objects downwards will inherit them.



Creating policies

Policies can be defined inside the [Azure Portal Policy page](#) under “Authoring / Definition” but we’ll do it as-code using Terraform today.

We’ll focus on the policy for forbidden regions here but the code for both [is available here](#).

```
resource "azurerm_policy_definition" "denied_regions" {
  name                = "Deny specific regions"
  policy_type         = "Custom"
  mode                = "All"
  display_name        = "Deny resources creation on certain specific regions"
  # management_group_id = azurerm_management_group.your_root_managementgroup.id # <- if you wa

  metadata = <<METADATA
    {
      "category": "General"
    }
  METADATA

  parameters = <<PARAMETERS
    {
      "deniedRegions": {
        "type": "Array",
        "metadata": {
          "displayName": "Regions to block",
          "description": "The Regions where it is not allowed to deploy resources"
        }
      }
    }
  PARAMETERS

  policy_rule = <<POLICY_RULE
    {
      "if": {
        "field": "location",
        "in": "[parameters('deniedRegions')]"
      },
      "then": {
        "effect": "deny"
      }
    }
  POLICY_RULE
}
```

Notes:

- As you can see this policy takes parameters (“deniedRegions” array) as input so it’s in fact a template.
- In the policies definition code we have set the “effect” to “deny” which means that resources meeting these specific conditions will be denied at creation-time.
- A smart way to implement policies is to set them in “audit” mode first to identify the potential impact and in “deny” mode to enforce them afterwards.

Assigning policies

Now that the policies are created, we need to assign them to our Management Group ("Authoring / Assignments in the Azure portal). Without an assignation a policy has no effect.

That's what we'll do with the following code (source available here):

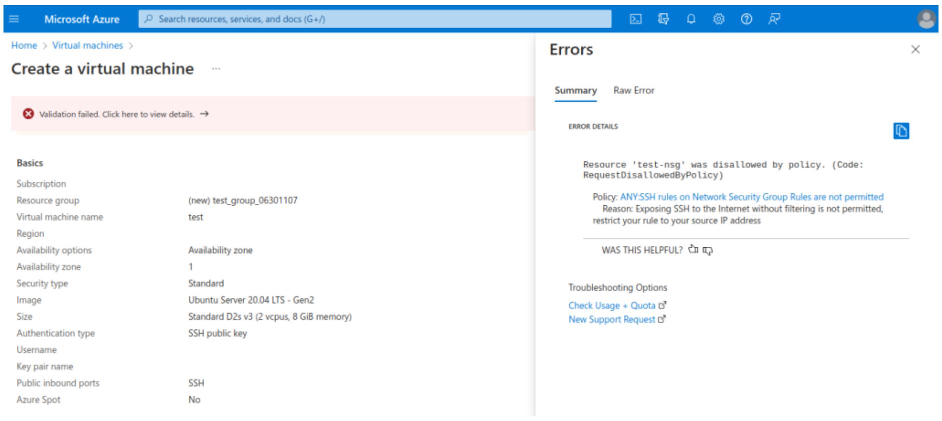
```
resource "azurerm_management_group_policy_assignment" "denied_regions" {
  name                     = "denied regions"
  display_name             = "Deploying resources in Brazil Southeast is not permitted"
  policy_definition_id     = azurerm_policy_definition.denied_regions.id
  # management_group_id = azurerm_management_group.your_root_managementgroup.id # <- if you want
  parameters = <<PARAMETERS
  {
    "deniedRegions": {
      "value": [ "Brazil Southeast" ]
    }
  }
  PARAMETERS
  non_compliance_message {
    content = "Deploying resources in Brazil Southeast is not permitted (due to costs reason)"
  }
}
```

Notes

- The "deniedRegions" key inside the parameters block, this is where we can set the variable that is applied inside the policy definition.
- Something also useful is the ability to define a non-compliance error message inside the policy assignment resource. This allows us to answer with a meaningful error message for the users when they deploy non-compliance resources, and it gives us the flexibility to define different error messages for the different assignments you may define (per subscription / teams, resources groups).

Testing the policies

Now that we have created and assigned our policies we can do a quick test from the Azure portal, by trying to create a new NSG rule that voids the policy or by deploying a resource in a forbidden region.



Create a virtual machine

Validation failed. Click here to view details. →

Basics

Subscription	
Resource group	(new) test_group_06301107
Virtual machine name	test
Region	
Availability options	Availability zone
Availability zone	1
Security type	Standard
Image	Ubuntu Server 20.04 LTS - Gen2
Size	Standard D2s v3 (2 vcpus, 8 GiB memory)
Authentication type	SSH public key
Username	
Key pair name	
Public inbound ports	SSH
Azure Spot	No

Errors

ERROR DETAILS

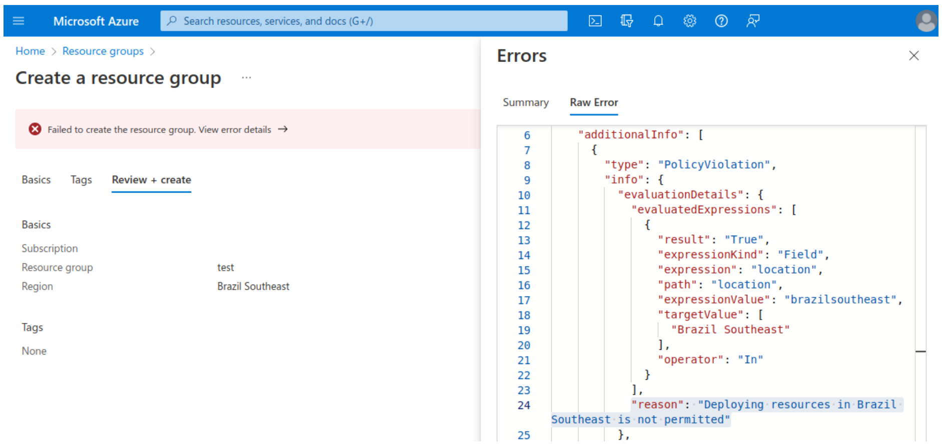
```
Resource 'test-nsg' was disallowed by policy. (Code:
RequestDisallowedByPolicy)
```

Policy: ANY:SSH rules on Network Security Group Rules are not permitted
Reason: Exposing SSH to the Internet without filtering is not permitted, restrict your rule to your source IP address

WAS THIS HELPFUL?

Troubleshooting Options

Deploying a NSG that voids the NSG policy



Create a resource group

Failed to create the resource group. View error details →

Basics

Subscription	
Resource group	test
Region	Brazil Southeast

Tags

None

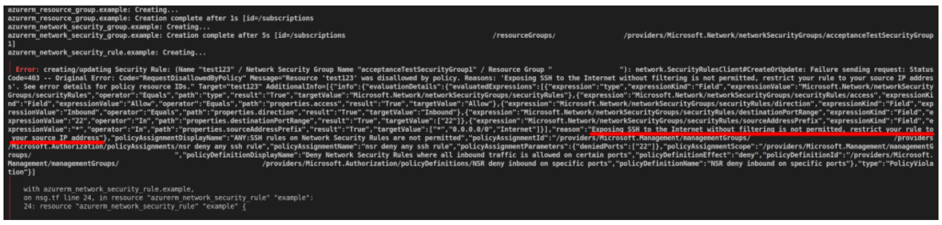
Errors

```
    "additionalInfo": [
      {
        "type": "PolicyViolation",
        "info": {
          "evaluationDetails": {
            "evaluatedExpressions": [
              {
                "result": "True",
                "expressionKind": "Field",
                "expression": "location",
                "path": "location",
                "expressionValue": "brazilsoutheast",
                "targetValue": [
                  "Brazil Southeast"
                ],
                "operator": "In"
              }
            ],
            "reason": "Deploying resources in Brazil
Southeast is not permitted"
          },
```

Deploying a resource in a forbidden region

We can see that our policies prevented the deployment of the resources, and that it did so during the evaluation process of the resource creation. This is quite powerful because it will warn users even before deploying the offending resources, instead of crashing with an error message at deployment-time.

The same will happen if we try to deploy the resource as-code, we will find the non-compliance error message inside the query response body.



```
azurerm_resource_group.example: Creating...
azurerm_resource_group.example: Creation complete after 1s [id=/subscriptions
azurerm_network_security_group.example: Creating...
azurerm_network_security_group.example: Creation complete after 5s [id=/subscriptions                    /resourceGroups/              /providers/Microsoft.Network/networkSecurityGroups/acceptanceTestSecurityGroup
1]
azurerm_network_security_rule.example: Creating...

 Error: creating/updating Security Rule: (Name "test123" / Network Security Group Name "acceptanceTestSecurityGroup1" / Resource Group "              "): network.SecurityRulesClient#CreateOrUpdate: Failure sending request: Status
Code=403 -- Original Error: Code="RequestDisallowedByPolicy" Message="Resource 'test123' was disallowed by policy. Reasons: 'Exposing SSH to the Internet without filtering is not permitted, restrict your rule to your source IP addres
s'. See error details for policy resource IDs." Target="test123" AdditionalInfo=[{"info":{"evaluationDetails":{"evaluatedExpressions":[{"expression":"type","expressionKind":"Field","expressionValue":"Microsoft.Network/networkSecurity
Groups/securityRules","operator":"Equals","path":"type","result":"True","targetValue":"Microsoft.Network/networkSecurityGroups/securityRules"},{"expression":"Microsoft.Network/networkSecurityGroups/securityRules/access","expressionKi
nd":"Field","expressionValue":"Allow","operator":"Equals","path":"properties.access","result":"True","targetValue":"Allow"},{"expression":"Microsoft.Network/networkSecurityGroups/securityRules/direction","expressionKind":"Field","exp
ressionValue":"Inbound","operator":"Equals","path":"properties.direction","result":"True","targetValue":"Inbound"},{"expression":"Microsoft.Network/networkSecurityGroups/securityRules/destinationPortRange","expressionKind":"Field","e
xpressionValue":"22","operator":"In","path":"properties.destinationPortRange","result":"True","targetValue":["22"]},{"expression":"Microsoft.Network/networkSecurityGroups/securityRules/sourceAddressPrefix","expressionKind":"Field","e
xpressionValue":"*","operator":"In","path":"properties.sourceAddressPrefix","result":"True","targetValue":["*","0.0.0.0/0","Internet"]}]},"reason":"Exposing SSH to the Internet without filtering is not permitted, restrict your rule to
 your source IP address"},"policyAssignmentDisplayName":"ANY:SSH rules on Network Security Rules are not permitted","policyAssignmentId":"/providers/Microsoft.Management/managementGroups/                    /providers
/Microsoft.Authorization/policyAssignments/nsr deny any ssh rule","policyAssignmentName":"nsr deny any ssh rule","policyAssignmentParameters":{"deniedPorts":["22"]},"policyAssignmentScope":"/providers/Microsoft.Management/managementG
roups/                    ","policyDefinitionDisplayName":"Deny Network Security Rules where all inbound traffic is allowed on certain ports","policyDefinitionEffect":"deny","policyDefinitionId":"/providers/Microsoft.
Management/managementGroups/                    /providers/Microsoft.Authorization/policyDefinitions/NSR deny inbound on specific ports","policyDefinitionName":"NSR deny inbound on specific ports"},"type":"PolicyViola
tion"}]

   with azurerm_network_security_rule.example,
   on nsg.tf line 24, in resource "azurerm_network_security_rule" "example":
   24: resource "azurerm_network_security_rule" "example" {
```

Managing exemptions

A good policy framework must also support exemptions for certain rare use cases. Azure Policy provides that out-of-the box. We won't cover exemptions in this blog post but they follow the same logic as policies assignments, you define a scope, a reason and optionally an expiration date for the exemption and that's it.

For more details check the [official documentation from Azure](#).

Unit-testing with Golang

I could have stopped here, but I wanted to go a bit further than that. These policies are working as expected, but I still had the following questions open in my mind. How to make sure that:

- The policy really blocks what it's supposed to?
- The policy behaves properly over time? For example, if some wrongly scoped exemptions are created?
- The policy does not block something it is not supposed to?
- The expected policy prevents deployment of a non-compliant resource in case there is overlapping policies?

Some of the questions above could be answered by the principle of least privilege, locking in the policies at the highest org-level and granting access only to certain super-admins. With infrastructure as-code, however, it is usually a service account that is used to push the changes on the platform and even with four-eye reviews, a mistake could happen.

To answer all these questions, I decided to implement unit tests. Initially I thought about doing them using Terraform code, but it was a bit cumbersome to catch and parse error messages and would mean that I had to wrap some bash or Golang around the Terraform code to achieve what I wanted to do.

Instead, I decided to implement the resource creation process using Golang code (leveraging the Azure SDK for Golang) and simply define unit tests using the standard testing package.

Here’s an extract of the testing scenarios for the forbidden regions (full code here).

```go
testRegionIn := func(region string) func() error {
        return func() error {
                _, err := CreateResourceGroup(ctx, cred, *azure, region)
                return err
        }
}

testCases := map[string]struct {
        test             func() error
        shouldError      bool
        linkedPolicyName string
}{
        "ResourceGroupCreation -> Brazil Southeast": {
                test:             testRegionIn("Brazil Southeast"),
                shouldError:      true,
                linkedPolicyName: "denied regions",
        },
        "ResourceGroupCreation -> Switzerland North": {
                test:        testRegionIn("Switzerland North"),
                shouldError: false,
        },
        "ResourceGroupCreation -> Switzerland West": {
                test:        testRegionIn("Switzerland West"),
                shouldError: false,
        },
```

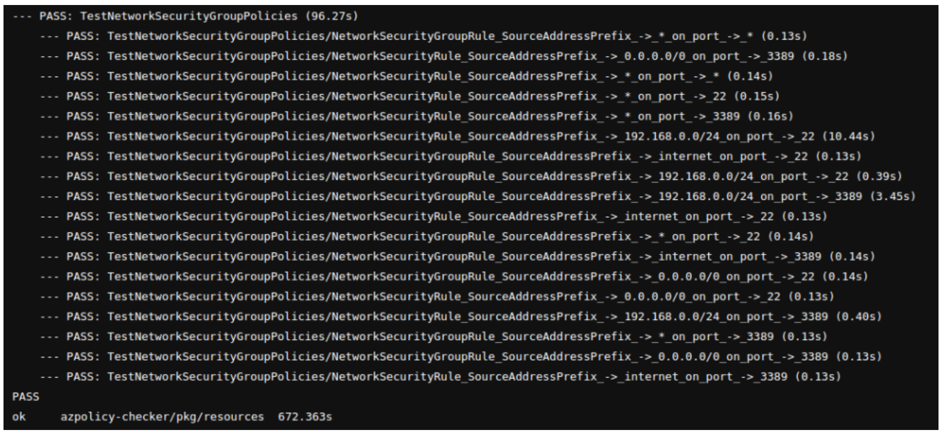
And here’s an extract of the logic to iterate over all the tests (full code here):

```go
for name, testCase := range testCases {
	t.Run(name, func(t *testing.T) {
		Cleanup(ctx, cred, *azure)

		err := testCase.test()

		switch {
		case err == nil && testCase.shouldError:
			t.Fatalf("Deployment was allowed but it should have been denied: %s", err)

		case err != nil && testCase.shouldError:
			if err := checkPolicy(err, testCase.linkedPolicyName); err != nil {
				t.Fatal(err)
			}

		case err != nil && !testCase.shouldError:
			t.Fatalf("Deployment was denied but it should have been allowed: %s", err)
		}
	})
}
```

Here's an extract of the testing scenarios for the forbidden NSGs (full code here).

```go
testCases := map[string]struct {
	test             func() error
	shouldError      bool
	linkedPolicyName string
}{
	"NetworkSecurityGroupRule SourceAddressPrefix -> * on port -> *": {
		test:             testSecurityGroupOn("*", "*"),
		shouldError:      true,
		linkedPolicyName: "nsg deny any any rule",
	},
	"NetworkSecurityGroupRule SourceAddressPrefix -> * on port -> 22": {
		test:             testSecurityGroupOn("*", "22"),
		shouldError:      true,
		linkedPolicyName: "nsg deny any ssh rule",
	},
	"NetworkSecurityGroupRule SourceAddressPrefix -> * on port -> 3389": {
		test:             testSecurityGroupOn("*", "3389"),
		shouldError:      true,
		linkedPolicyName: "nsg deny any rdp rule",
	},
	"NetworkSecurityGroupRule SourceAddressPrefix -> internet on port -> 22": {
		test:             testSecurityGroupOn("internet", "22"),
		shouldError:      true,
		linkedPolicyName: "nsg deny any ssh rule",
	},
	"NetworkSecurityGroupRule SourceAddressPrefix -> internet on port -> 3389": {
		test:             testSecurityGroupOn("internet", "3389"),
		shouldError:      true,
		linkedPolicyName: "nsg deny any rdp rule",
	},
```

And here is how it looks like when being executed (here the NSG tests):



```
--- PASS: TestNetworkSecurityGroupPolicies (96.27s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_*_on_port_->_* (0.13s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_0.0.0.0/0_on_port_->_3389 (0.18s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_*_on_port_->_* (0.14s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_*_on_port_->_22 (0.15s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_*_on_port_->_3389 (0.16s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_192.168.0.0/24_on_port_->_22 (10.44s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_internet_on_port_->_22 (0.13s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_192.168.0.0/24_on_port_->_22 (0.39s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_192.168.0.0/24_on_port_->_3389 (3.45s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_internet_on_port_->_22 (0.13s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_*_on_port_->_22 (0.14s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_internet_on_port_->_3389 (0.14s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_0.0.0.0/0_on_port_->_22 (0.14s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_0.0.0.0/0_on_port_->_22 (0.13s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_192.168.0.0/24_on_port_->_3389 (0.40s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_*_on_port_->_3389 (0.13s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityGroupRule_SourceAddressPrefix_->_0.0.0.0/0_on_port_->_3389 (0.13s)
    --- PASS: TestNetworkSecurityGroupPolicies/NetworkSecurityRule_SourceAddressPrefix_->_internet_on_port_->_3389 (0.13s)
PASS
ok      azpolicy-checker/pkg/resources  672.363s
```

We can see that all the tests successfully passed as expected.

Wrapping up

In this blog post we saw how to improve our Cloud-native security by implementing guardrails as-code leveraging Azure Policies.

We also saw how to make sure that these policies remain consistent over-time by coding some unit-tests in Golang.

It is understood that all the Terraform code and the unit tests must be part of steps in a Continuous-Integration/Continuous-Deployment (CI/CD) pipeline.

All the examples above are available on the following Github repository.

Finally if you have questions about cloud-native security or security in general feel free to reach out to us !

-Romain