Automated Policy for developers using CI/CD (Terraform) tools
Infrastructure as a code is a core element of today’s CI/CD pipeline and led to the emergence of Continuous Configuration Automation (CCA) tools, such as Terraform, a leading declarative push CCA solution launched by HashiCorp in 2014.

Infrastructure-as-code benefits are undeniable:

- It simplifies compute resource provisioning by leveraging automation and scalability.
- Through a simple, easy to use, declarative language, it enables you to simply express your infrastructure desired state - and to have it be automatically configured.
- It makes infrastructure deployment consistent and predictable, which minimizes human errors.
- It codifies institutional knowledge, which reduces future risk.
- It saves enormous time and resources.
Yet, any emerging technology, especially when used as the backbone of innumerable products and services, creates new “supply chain” challenges. Infrastructure-as-a-code solutions are developed and designed to provision functional infrastructure resources and streamline CI/CD. Security is an afterthought, and unfortunately, not prioritized nearly as much as it should be.

For example, Terraform offers ease-of-use and flexibility through reusable modules and registries made available by Terraform users who created them for their own use and want to help the terraform community. However, a study run in the last quarter of 2020 shows that up 44% of the 2,600 modules for Amazon Web Services, Azure, and Google Cloud support were misconfigured when assessed how they match up against CIS benchmarks.

These findings are consistent across modules and registries for all major cloud service providers.

As infrastructure-as-a-code is still a relatively new technology with distinct methodology and off-the-shelf code-level static analysis, traditional cloud security tools do not yet fully support it. This leaves gaps in coverage and unmitigated risk.
Cisco Secure Cloud-Native is already preventing damages from most of these misconfigurations but was still missing an important tool to eradicate the information gap between developers at the build stage and DevOps at the deployment stage. These gaps complicated the efficient management of pod-specific cloud-native security policies risking security gaps in newly deployed service, creating unwanted deployment delays, or generating broken or insecure communication channels at deployment.

The need to plug that gap and solve the resulting deployment delays are what led Cisco to work on developing a plugin that automates the policy creation to solve these issues and provide cloud-native security from the get-go.
Cisco Secure Cloud-Native

When it comes to infrastructure-as-a-code, though Cisco’s CI/CD security level is quite advanced, it still had some challenges to overcome, so we’ve developed a plugin for Terraform.

To understand what that plugin does, let’s have a look at the current security gap in the CI/CD pipeline.
The CI/CD security hole

To understand where the CI/CD security gap is when using Terraform, you need to understand what a Kubernetes Security Context is.

The Kubernetes Security Context is what the pod requests. Pod Security Policies (PSP) needed to be calibrated for each pod to enforce the security context and maximize both flexibility and security. PSPs are now being deprecated, but Kubernetes is developing a PSP substitute covering key use cases in a simpler and more sustainable way.

The crux of the problem lies here. Increasing flexibility decreases security, and increasing security decreases flexibility.
Example

Deploying this Kubernetes deployment will create a workload with AllowPrivilegeEscalation:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
labels:
  app: nginx
spec:
  replicas: 3
selector:
  matchLabels:
    app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
```

Creating a psp profile that disables this option will fix this security hole
To find the optimal balance, the DevOps personnel configuring pod policies needs to know precisely what level of privileges to attach to the service and define connection policies by granularly allowing or restricting connections between workloads, environments, and internal or external databases, applications, and other source and destination entities.
What traditionally happens?

In the CI/CD classic flow, the basic steps are deceptively simple. Launching a new service in, for example, a Kubernetes environment, can be summarily schematized in four steps:

1. A developer writes the code for that new service and the Kubernetes deployment files.

2. The developer pushes it to the master nodeDevOps updates the new service’s security policies.

3. The new service is deployed, and the new pod appears in the production environment.
These steps work well for small projects where the developer and the DevOps are one and the same person.

In a small team, with a direct line of communication, Cisco Secure Cloud-Native will flag the workload as new and requiring review. The user can receive a notification in Slack, Cisco teams, or other tools to alert them. The DevOps, DevSecOps, or security analyst managing Cisco Secure Cloud-Native should ideally take the time to circle back to the developer who produced the new workload. They will clarify the source and destination entities requiring communication before approving and deploying the code.
This verification process creates delays as the DevOps, DevSecOps, or the security analyst needs to extract the information from the developer, who is not always available. Those delays create tensions with management eager to see the product launched already.

Caught in the middle, the DevOps, DevSecOps, or security analyst might opt to insert the proper context instead of waiting for the developer’s answer. This leap could increase the risk of introducing vulnerabilities or creating a breakdown in communication between the workload and necessary source or destination entities such as database, services or other.

At a small company, identifying and contacting the relevant developer should take a limited time and be manageable, albeit with inefficiencies.

But if you imagine a large company with 100 developers each producing 10 new services a week and pushing them to production, the DevOps, DevSecOps, or security analyst department implementing security cannot humanely be expected to identify and contact each developer individually.
They cannot be expected to obtain a clear idea of each newly created pod’s context, required source and destination entities.

Without detailed information about each new service’s full context, DevOps are configuring policies that:

- Are vulnerable to unwanted communication due to too lax policies
- Risk leading to blocked communication due to too tightly configured permission

The alternative is creating considerable deployment delays to collect the necessary context information.

Even if no resulting vulnerability is ever leveraged to penetrate the environment, this communication gap results in high operating costs, unnecessary stress for DevSec-Ops, and tensions or blame-shifting between departments.
Terraform Provider

Terraform Provider is a plug-in solution designed to automate the necessary information exchange between developers and DevSecOps/DevOps, even after Kubernetes PSP complete deprecation.
Cloud Native Security Automated Policy Advisor Solution

Integrated with Cisco Cloud Native Security, the Terraform Provider plugin solves the vast majority of the context and communication information gap between development and deployment and enables the automation of custom security policies for deployment without human intervention.

Terraform Provider is an Automated Policy Advisor Solution that works in Cisco Secure Cloud-Native environments, where the Cisco Secure Cloud-Native’s controller checks if workloads abide by environment-specific deployment policy rules.

In short, it embeds crucial elements of CI/CD security for cloud deployment at the build stage, enabling secure automated deployment.
When you use the Terraform Provider plugin to create or modify a cluster in Cisco Secure Cloud-Native, this is what happens:

1. Cisco Secure Cloud-Native deployment policies add the deployment rules applying to select workloads and environments, and select the properties or actions that the rules will perform.

To add the cluster

```terraform
provider "portshift" {
  access_key = "**********
  secret_key = "**********"
}

resource "portshift_k8_cluster" "myCluster" {
  kubernetes_cluster_context = "clusterContext"
  portshift_cluster_name = "myPortshiftCluster"
  ci_image_validation = "false"
  cd_pod_template = "false"
  connections_control = "true"
  multi_cluster_communication_support = "false"
  inspect_incoming_cluster_connections = "false"
  fail_close = "false"
  persistent_storage = "false"
  external_https_proxy = ""
}
```
Example of adding a connection rule

```
resource "portshift_connection_rule" "name in terraform" {
  rule_name = "name in portshift"
  source_by_ip_range {
    ips = ["192.168.1.0/24"]
  }
  destination_by_address_ip_range {
    ips = ["192.168.1.0/24"]
  }
}
```

When run you can see the rules in the connection policy
2. When a developer writes deployment files to attach to a newly created workload, he adds a Terraform Provider object to the deployment files. By granularly defining all source and destination entities, this new object allows the DevOps, DevSecOps, or security analyst in charge to know exactly what the new workload’s service context and communication requirements are, so they can calibrate the policies without needing to circle back to the developer and without having to guess anything.
The required source entities for the new service can be listed by the following parameters:

- IP (including the option to define an ip_range)
- Selected external sources
- Pod name
- Pod label
- Vulnerability severity level
- Any pod
And the destination entities can be defined by:

- Address IP range (including the option to define an ip_range)
- Address Domain
- Selected external sources
- Pod name
- Pod label
- Vulnerability severity level
- Any pod
When the Cisco Secure Cloud-Native controller reads the Terraform Provider object, all connections affected by a rule are displayed, giving an instant birds-eye view of the downstream effect of a rule modification on the application running. This enables administrators to set and centralize the management of operational policies, minimizing conflicting rules and improving Kubernetes-orchestrated applications’ security profile.

Depending on the Cisco Secure Cloud-Native enforcement levels, advisory, soft-mandatory, or hard-mandatory required policy rules conveyed through Terraform Provider, the tool will trigger a warning, an alarm, or a rejection of the workload. This draws the DevOps/DevSecOps attention to the policy conflict while providing a full context of the communication requirement for the offending workload.

In full possession of all relevant information, the DevOps/DevSecOps can then either tweak the policy rules to match the workload requirements or send the workload back to the developer with specific information of what connection source or destination is creating an unacceptable security risk.
How Can You Access Terraform Provider?

Included in Cisco Secure Cloud-Native, Terraform Provider adds, modifies, or removes Cisco Secure Cloud-Native to Kubernetes Clusters, and uses Terraform files to create or modify Cisco Secure Cloud-Native connection rules.

To access Terraform Provider, you need a Cisco Secure Cloud-Native Service account and Access Keys. In addition to this your host machine has to be Linux with kubectl installed.

Your Kubernetes cluster must have Cisco Secure Cloud-Native as the defined context in config files.

Contact us for more information and to try our latest tool out