HashiCorp Terraform

Writing and Testing Sentinel Policies for Terraform

Version 3.0

by Roger Berlind

Introduction

This guide provides instruction for writing and testing Sentinel policies for Terraform. Sentinel allows customers to implement governance policies as code in the same way that Terraform allows them to implement infrastructure as code. Sentinel policies define rules that restrict the provisioning of resources by Terraform configurations. Terraform enforces Sentinel policies between the plan and apply phases of a run, preventing out of policy infrastructure from being provisioned. Unless overridden by an authorized user, only plans that pass all Sentinel policies checked against them are allowed to proceed to the apply step.

This guide discusses the types of Sentinel policies in Terraform and lays out a general methodology for writing and testing Sentinel policies for Terraform. It also covers useful Sentinel operators, functions, and concepts, and how to use the Sentinel CLI to test your policies. Finally, it references examples and exercises from the Sentinel for Terraform Workshop that you can study independently in order to learn how to write and test Sentinel policies for Terraform.

Types of Sentinel Policies for Terraform

There are essentially four types of Sentinel policies for Terraform which correspond to these four Sentinel imports: tfplan, tfconfig, tfstate, and tfrun. The first and most common type of policy uses the tfplan import to restrict attributes of specific resources or data sources. The second type uses the tfconfig import to restrict the configuration of Terraform modules, variables, resources, data sources, providers, provisioners, and outputs. The third type uses the tfstate import to check whether any previously provisioned resources, data sources, or outputs have attribute values that are no longer allowed by your governance policies. The fourth type uses the tfrun import to check workspace and run metadata and whether cost estimates for planned resources are within limits.

This guide focuses primarily on the first type of Sentinel policy that uses the tfplan import, but we do cover the other imports in later sections.

Methods for Testing Terraform Sentinel Policies

You can use three different methods for testing your Terraform Sentinel policies:

1. You can manually test policies against actual Terraform code by using the Terraform UI or the Terraform CLI with the remote backend to trigger runs against workspaces that use that Terraform code.
2. You can execute automated tests against actual Terraform code by using the Terraform API.
3. You can use the Sentinel CLI with mocks generated from Terraform plans.

In the first two methods, you are executing plans against your Terraform code and then testing the Sentinel policies against the generated plans. In the third method, you only use Terraform to generate your mocks. Because this method employs simulated testing, we recommend

deploying policies to a Terraform server and running final testing against actual Terraform plans as well.

While the first method is sometimes easier for new Terraform users to start with, it does have some disadvantages:

1. It is a manual method rather than an automated method.
2. Each test will take longer than if you used the Sentinel CLI since Terraform has to run a plan against your workspace before it even invokes any Sentinel policies.
3. Unless you use Terraform policy sets carefully, you might end up running multiple policies for each test even though you only care about the one you are testing.
4. If you use the Terraform UI, all the runs you do to test your policy will end up in the histories of your workspaces and you will need to discard each run you do that passes your policies.

Using the Terraform CLI with the remote backend instead of the Terraform UI avoids the fourth problem because it runs speculative plans which cannot be applied and do not show up in the workspace history in the Terraform UI. This means that you do not need to discard any runs and won't have a long history of them in your workspaces. Additionally, if you use the Terraform CLI, you do not need to put your Terraform code in or link your workspaces to VCS repositories.

This guide will focus on the third method described above.

Basic Methodology for Restricting Resources with Sentinel

In this section, we lay out a basic methodology for restricting specific attributes of specific Terraform resources and data sources in Sentinel policies. We initially focus on resources but will cover data sources later.

Before proceeding, we want to make an important point about the scope of these Sentinel policies within Terraform: They are intended to ensure that specific attributes of resources are included in all Terraform code that creates those resources and have specific values. They are not intended to ensure that the specified values are actually valid. In fact, invalid values will cause the plan or apply to fail since Terraform and its providers ensure that attribute values are legitimate. Together, Sentinel and Terraform do ensure that all resources have attribute values that are both compliant with governance policies and valid.

Documentation You will Need:

In general, when you write a Sentinel policy to restrict attributes of Terraform resources or data sources, you should have the following documents at hand:

1. The tfplan import documentation.
2. The Sentinel Language documentation.
3. The third-generation Sentinel examples from the terraform-guides repository, which are organized by cloud (AWS, Azure, GCP, and VMware).
4. The Terraform documentation for the resource or data source you wish to restrict.

5. Documentation from the cloud service or other technology vendor about the resource that is being created.

For example, if you wanted to restrict certain attributes of virtual machines in the Google Cloud Platform, you would visit the Terraform Providers page, click the Google Cloud Platform link, and then search the list of data sources and resources for the google_compute_instance resource. Having this page in front of you will help you determine which attributes you can use for the google_compute_instance resource in your Sentinel policy.

You might also need to refer to Google's own documentation about google compute instances. Googling "google compute instance" will lead you to Google's Virtual Machine Instances page. You might also need to consult Google's documentation about the attributes you want to restrict. In our case, we will find it useful to consult Google's Machine Types page.

Note that resource and data source documentation pages in the Terraform documentation list both "arguments" and "attributes". From Sentinel's point of view, these are the same thing, so we will only talk about "attributes" below. However, when you read the Terraform provider documentation for a resource or a data source, you will want to look at both its arguments and its attributes. However, it is important to realize that the values of some attributes are computed during the apply step of a Terraform run and cannot be evaluated by Sentinel.

Sometimes, you might be asked to restrict some entity without being told the specific Terraform resource that implements it. Since some providers offer more than one resource that provides similar functionality, you might need to search the provider documentation with multiple keywords to find all relevant resources. The AWS Provider, for example, has "aws_lb","aws_alb", and "aws_elb" resources that all create load balancers. So, if you want to restrict load balancers in AWS, you would probably need to restrict all of these AWS resources.

Some Prerequisites:

You can learn how to write and test Sentinel policies for Terraform by using the hands-on labs within the Sentinel for Terraform Workshop which uses the Sentinel CLI. Using the workshop labs only requires a browser with access to the internet. If you use the workshop labs, you can ignore all instructions below about downloading and installing the Sentinel CLI, writing Terraform code, generating mocks, and so on. But reading the rest of this guide will give you more insight into how you would write and test Sentinel policies for Terraform outside of the workshop labs.

If you do want to use the methodology described in this guide against actual Terraform code, you will also need to satisfy the following prerequisites:

1. You should either have an account on the public Terraform Cloud server at https://app.terraform.io or access to a (private) Terraform Enterprise (TFE) server.
2. You will need to belong to the owners team within an organization on the Terraform server you are using or have the Manage Policies organization permission for the organization.

3. If you want to create Sentinel policies that test resources in a public cloud and actually run your policies on a Terraform server, you will need to have an account within that cloud. If you want to test the restrict-ec2-instance-type policy described below against actual Terraform code, you will need an AWS account and AWS keys. If you want to test the restrict-gce-machine-type policy described below against actual Terraform code, you will need a Google Cloud Platform account and a Google Cloud service account file containing your GCP credentials. The Getting Started with Authentication document from Google will show you how to create a service account and download a credentials file for it. You do not need cloud accounts in order to write and test policies with the Sentinel CLI, but you will need them in order to run plans in Terraform workspaces and generate mocks from those plans.
4. You will need to download and install the Sentinel CLI.

Downloading and Installing the Sentinel CLI:

Download the Sentinel CLI for your OS from the Sentinel Downloads page. We used version 0.15.3 which was the most current version at the time we wrote the Sentinel for Terraform v3 workshop. You should probably download the most recent version.

After downloading Sentinel, unzip the package which contains a single binary called "sentinel". Then add that binary to your path. On a Linux or Mac, edit ~/.profile, ~/.bash_profile, or ~/.bashrc and add a new line like "export PATH=$PATH:< path_to_sentinel >". On Windows, navigate to Control Panel -> System -> System settings -> Environment Variables and then add ";<path_to_sentinel>" to the end of the PATH environment variable. Alternatively, you could temporarily add Sentinel to your path in a Windows command shell by running "set PATH=%PATH%;<path_to_sentinel>".

Outline of the Methodology:

Here is an outline of the steps you will perform while creating and testing your Sentinel policies. Below the outline, we provide details about each step. This covers the third method listed above: testing policies with the Sentinel CLI after generating a mock from a Terraform plan.

1. Write a Terraform configuration that creates the resource your policy will restrict.
2. Create a Terraform workspace that uses your Terraform configuration.
3. Run a plan against your workspace using the remote backend.
4. Generate mocks against your plan in the Terraform UI.
5. Write a new Sentinel policy.
6. Test your Sentinel policy with the Sentinel CLI
7. Revise your policy and test cases until they all pass.
8. Deploy your policy to an organization on a Terraform server.

We will now illustrate how you can follow these steps to create and test a Sentinel policy to restrict the size of Google compute instances by borrowing code from the third-generation

restrict-ec2-instance-type policy, which restricts the size of AWS EC2 instances. Basing our new policy on this one makes sense since Google compute instances and EC2 instances are both virtual machines (VMs) in their respective clouds. These are frequently used Sentinel policies because they help reduce costs by controlling the size of VMs in clouds. The repository with the AWS policy actually contains a corresponding GCP policy, but we will walk you through the process of re-creating it as if it did not exist.

Note: We wrote earlier versions of this guide before creating the Sentinel for Terraform Workshop. At that time, we expected readers to actually execute the 8 steps given below and therefore provided a lot of details. However, we would now recommend that you work your way through the exercises of the workshop instead of trying to implement the 8 steps below unless you really want to test out the policies we describe on an actual Terraform Cloud or Terraform Enterprise server.

Step 1: Write a Terraform Configuration that Creates the Resource:

Let's write a Terraform configuration that will create a google_compute_instance resource so that we can test a Sentinel policy against it. We will use a Terraform variable called machine_type to set the machine_type attribute of the google_compute_instance with different values in pass and fail mocks. When we set it to "n1-standard-2" or "n1-standard-4", the policy should pass; but if we set it to "n1-standard-8" (or anything else), it should fail.

We don't have to look very far to find an example for creating a google_compute_instance; there is an example right inside the documentation for it.

Examples like this in the Terraform provider documentation can quickly be modified to test your Sentinel policies. You might need to add additional attributes if your policy is testing their values and add variables to represent them. You might also want to modify the code to create multiple instances of the resource, including some that would pass and some that would fail your policy. You might also need to add code to configure authentication credentials for Terraform providers.

Here is some Terraform code you can use to test the restrict-gce-machine-type policy that we will create in Step 5:

```
variable "gcp_project" {
  description = "GCP project name"
}

variable "machine_type" {
  description = "GCP machine type"
  default = "n1-standard-1"
}

variable "instance_name" {
  description = "GCP instance name"
```

```
  default = "demo"
}

variable "image" {
  description = "GCP image"
  default = "debian-cloud/debian-9"
}

provider "google" {
  project     = "${var.gcp_project}"
  region      = "us-east1"
}

resource "google_compute_instance" "demo" {
  name         = "${var.instance_name}"
  machine_type = "${var.machine_type}"
  zone         = "us-east1-b"

  boot_disk {
    initialize_params {
      image = "${var.image}"
    }
  }

  network_interface {
    network = "default"

    access_config {
      // Ephemeral IP
    }
  }

}
```

Put this code in a single main.tf file and add the following code to a separate backend.tf file:

```
terraform {
  backend "remote" {
    hostname = "app.terraform.io"
    organization = "<your_org>"

    workspaces {
      name = "<your_workspace>"
    }
  }
}
```

Be sure to set <your_org> to the name of your organization on the Terraform server you are using and <your_workspace> to the name of the workspace you will create in that organization in Step 2. If you are using a private Terraform server, change the hostname field to its URL. Additionally, be sure to generate a user API token for yourself and list it in your Terraform CLI configuration file (.terraformrc on Mac/Linux or terraform.rc on Windows).

Step 2: Create a Workspace that Uses Your Terraform Configuration:

After writing your Terraform configuration in Step 1, you need to create a workspace for it. To do this, follow the instructions for creating workspaces. Select the "CLI-driven workflow" option for your workspace so that it is not linked to a VCS repository.

After creating your workspace, be sure to set the gcp_project Terraform variable on the Variables tab of your workspace. The gcp_project variable should be set to the ID of the project in your GCP account in which you want to create the VM; that is typically but not always the name of the project. Create an environment variable on the same tab called GOOGLE_CREDENTIALS with the contents of your local copy of the Google Cloud service account file containing your GCP credentials after stripping out all line breaks from it. For more on this, see the Google Provider's Configuration Reference page and the GCP documentation it links to. Please be sure to mark your GOOGLE_CREDENTIALS variable as sensitive so that other people cannot see or copy what you paste into it.

Please also visit the General Settings tab of your workspace and set the Terraform Version to the version you wish to use with the workspace. The version you set here (rather than your local version) will determine the version of the mocks you generate.

Step 3: Run a Plan Against Your Workspace:

After creating your workspace, run a plan against it using the remote backend so that you can generate mocks for it in the next step. You can do this with either Terraform 0.12.x or 0.13.x. On your local machine, navigate to the directory containing the main.tf and backend.tf files you created in Step 1. Then run `terraform init` to initialize your configuration and `terraform plan` to run a plan. If your backend.tf file is configured correctly and pointing at the workspace you created in Step 2, you will see a link to the run containing the plan in the Terraform UI. Copy this link and paste it into a browser so you can see the plan and generate mocks in the UI.

Step 4: Generate Mocks Against Your Plan:

After running a plan and navigating to it in the Terraform UI, click on the Plan step within the run to expand it if it is not already expanded. You can then generate tfplan, tfconfig, and tfstate mocks from the plan by clicking on the "Download Sentinel mocks" button. These mocks correspond to the four Sentinel imports and can be copied and edited to simulate the Sentinel data produced from Terraform plans in during actual runs in Terraform. Clicking the button will download a run-<id>-sentinel-mocks.tar.gz file. On a Mac, you can double-click this file to extract the mocks. On a Windows machine, you can use WinZip or other utility to extract them. On Linux, you can run `tar xvzf <file>` to extract them. You will find multiple files

including mock-tfconfig.sentinel, mock-tfplan.sentinel, and mock-tfstate.sentinel (in two versions) and mock-tfrun.sentinel. We will primarily use version 2 of the mock-tfplan.sentinel file.

Note: Currently the version 1 mocks do not have "v1" in their names while the version 2 mocks have "v2" in their names, but we expect that to change in the near future at which point the version 1 mocks will have "v1" in their names and the version 2 mocks will not have versions indicated. Whenever we refer to a file like "mock-tfplan.sentinel" below, we mean the version 2 mock for the "tfplan" import.

Step 5: Write a New Sentinel Policy:

On your local machine, you will now write a new Sentinel policy that will use the mock-tfplan.sentinel mock you generated in Step 4. Be sure that you already downloaded and installed the Sentinel CLI and that the sentinel binary is in your path.

You can use any text editor you want to edit your policy but be sure to save the file with the ".sentinel" extension. We suggest using a name for the policy that indicates what it does and indicates the type of resource and particular attributes being restricted. For example, we could call the new policy "restrict-gce-machine-type", which indicates the resource and attribute being restricted. Of course, if you are restricting many attributes of a resource or restricting multiple resources, then using a policy name like this might not be feasible. However, it is generally best to limit each policy to a single resource unless they are very similar. For example, you might create a single policy to limit the aws_lb and aws_elb resources since they both create AWS load balancers.

The easiest way to create the code for a new Sentinel policy is to copy code from an existing policy in the terraform-guides repository that also restricts some resource and then make changes to the copied code. If possible, pick a policy that restricts the same resource you wish to restrict. If you can't find one that does that, it can be useful to pick one that uses a similar resource even if that resource belongs to a different cloud.

As a reminder we will be copying code from the third-generation restrict-ec2-instance-type policy. Like many other Sentinel policies in the terraform-guides repository, this policy uses several common functions including find_resources and filter_attribute_not_in_list which are both from the tfplan-functions Sentinel module. These are parameterized functions that can be reused in many policies. You will therefore not have to change very much code when creating a new policy from an existing one.

Using Sentinel Modules allows us to define functions once and then reuse them in multiple policies. Using them also keeps policies much shorter and easier to understand.

Here is the entire policy:

```
# This policy uses the Sentinel tfplan/v2 import to require that
# all EC2 instances have instance types from an allowed list
```

```
# Import common-functions/tfplan-functions/tfplan-functions.sentinel
# with alias "plan"
import "tfplan-functions" as plan

# Allowed EC2 Instance Types
# Include "null" to allow missing or computed values
allowed_types = ["t2.small", "t2.medium", "t2.large"]

# Get all EC2 instances
allEC2Instances = plan.find_resources("aws_instance")

# Filter to EC2 instances with violations
# Warnings will be printed for all violations since the last parameter
is true
violatingEC2Instances =
plan.filter_attribute_not_in_list(allEC2Instances,
                        "instance_type", allowed_types, true)

# Count violations
violations = length(violatingEC2Instances["messages"])

# Main rule
main = rule {
  violations is 0
}
```

The entire policy is only 26 lines even with extensive comments. The number of lines of actual code in the policy is only 8.

Each Sentinel policy intended to restrict attributes of resources or data sources will import the tfplan-functions module by including this line at the top. Note that we assign the import the `plan` alias to keep lines that call its functions shorter.

```
import "tfplan-functions" as plan
```

The policy next defines a list of allowed EC2 instance types:

```
allowed_types = ["t2.small", "t2.medium", "t2.large"]
```

It then calls the `find_resources` function of the tfplan-functions import, using the `plan` alias and passing in the string "aws_instance" to indicate that the policy wants the function to return all instances of the aws_instance resource.

```
allEC2Instances = plan.find_resources("aws_instance")
```

The `find_resources` function retrieves all instances of a specified resource from all Terraform modules for the current plan. Every Terraform configuration has at least one module, namely the root module which contains the top-level Terraform code against which you run the plan and apply commands. Retrieving instances from all modules is very important because your Sentinel policy is useless if you only restrict the creation of the resource in the root module.

Here is the code of the `find_resources` function with minor formatting changes:

```
find_resources = func(type) {
  resources = filter tfplan.resource_changes as address, rc {
  rc.type is type and
  rc.mode is "managed" and
  (rc.change.actions contains "create" or rc.change.actions contains "update")
  }

  return resources
}
```

This function accepts a single parameter called `type` which is a string that specifies the type of resource we want to find. Since this uses the v2 tfplan import, all resources of the specified type from all modules will be returned in a flat map. So, we do not need to do multiple loops as was the case for the v1 tfplan import. Instead, we simply use the Sentinel filter expression to filter out the resource changes of the desired type that are also managed (which means they are not data sources) and which are being created or updated. We impose the last restriction to avoid applying the policy to resources that are being destroyed.

Note that the discovered resources of the desired type will be indexed by their full address.

The restrict-ec2-instance-type policy also calls the `filter_attribute_not_in_list` function which filters a collection of resources to those with a specified attribute that has values which are not in a given list. Here is the call to this function in the policy:

```
violatingEC2Instances = plan.filter_attribute_not_in_list(
    allEC2Instances, "instance_type", allowed_types, true)
```

Note that we are passing in the `allEC2Instances` collection returned by the `find_resources` function and the `allowed_types` list. We set the fourth parameter, `prtmsg`, to `true` so that the function will print all violations that it finds.

Here is the code for the function itself:

```
filter_attribute_not_in_list = func(resources, attr, allowed, prtmsg) {
  violators = {}
  messages = {}
  for resources as address, rc {
    # Evaluate the value (v) of the attribute
```

```
    v = evaluate_attribute(rc, attr) else null
    # Convert null to "null"
    if v is null {
      v = "null"
    }
    # Check if the value is not in the allowed list
    if v not in allowed {
      # Add the resource and a warning message to the violators list
      message = to_string(address) + " has " + to_string(attr) + " with value
" +
                to_string(v) + " that is not in the allowed list: " +
                to_string(allowed)
      violators[address] = rc
      messages[address] = message
      if prtmsg {
        print(message)
      }
    } // end if
  } // end for
  return {"resources":violators,"messages":messages}
}
```

This function has four arguments: `resources`, `attr`, `allowed`, and `prtmsg`. The first is the type of the collection of resources to be filtered, the second is the attribute that is being checked, the third is the list of allowed values for that attribute, and the fourth is a boolean indicating whether violation messages should be printed as violations are found. (If set to `false`, the violation messages can still be printed later.)

The function defines two maps called `violators` and `messages` which the function will return. The function iterates over all resource instances passed to it and calls a third function, `evaluate_attribute`, to evaluate the specified attribute for the current resource. If the attribute value is not in the list, the resource is added to the `violators` map and a violation message is added to the `messages` map. This allows the function to print and return violation messages for all resource instances that have violations.

We don't show the code for the `evaluate_attribute` function here, but it is a very important function in the tfplan-functions and tfstate-functions Sentinel modules since it can evaluate any attribute of any Terraform resource or data source even if the attribute is deeply nested.

The policy next counts the number of violation messages returned by the filter function:

```
violations = length(violatingEC2Instances["messages"])
```

Here is the `main` rule that actually restricts the size of the aws_instance resources:

```
main = rule {
  violations is 0
}
```

Note that the rule simply checks that the number of violations is 0.

Sentinel rules must return a boolean value, `true` or `false`. While a policy can define separate rules for each condition and then combine them with a compound boolean expression in the main rule, we prefer to only use a main rule to reduce the amount of default Sentinel output generated.

Now, let's think about how we can modify the policy to restrict Google compute instances instead of AWS EC2 instances. Without even looking at the documentation for the google_compute_instance resource, we can reasonably expect that we might need to change the items in the `allowed_types` list and the specific resource and attribute passed to the functions. When you do look at that documentation, you will find that one of its required attributes is machine_type. Since there is no top-level size, instance_size, or instance_type attribute for this resource, it's quite likely that machine_type is the attribute we want to use. We can verify that by checking Google's documentation.

When we discussed useful documentation that you will need for creating Sentinel policies, we had this example in mind and referred to Google's documentation about Virtual Machine Instances and Machine Types. If you look at the latter, you'll see that the machine type of a Google virtual machine determines the amount of resources available to the VM. You'll also see a list of predefined machine types. The smallest standard machine types are "n1-standard-1", "n1-standard-2", and "n1-standard-4". So, we could modify our `allowed_types` list to include those three types:

```
allowed_types = ["n1-standard-1", "n1-standard-2", "n1-standard-4"]
```

We kept the name of the list since it works equally well for restricting Google Virtual Machine machine types as for AWS EC2 instance types.

Now that we know that we want to restrict the `machine_type` attribute of the `google_compute_instance` resource, we can modify our function calls to look like this:

```
allGCEInstances = plan.find_resources("google_compute_instance")
```

and

```
violatingGCEInstances = plan.filter_attribute_not_in_list(
           allGCEInstances,"machine_type", allowed_types, true)
```

Note that we have changed `aws_instance` to `google_compute_instance` and changed `instance_type` to `machine_type`. We also changed the names of the collections from `allEC2Instances` and `violatingEC2Instances` to `allGCEInstances` and `violatingGCEInstances` respectively.

That completes the modifications to our policy, which should be called "restrict-gce-machine-type.sentinel" and should look like this:

```
# This policy uses the Sentinel tfplan/v2 import to require that
# all GCE instances have machine types from an allowed list

# Import common-functions/tfplan-functions/tfplan-functions.sentinel
# with alias "plan"
import "tfplan-functions" as plan

# Allowed GCE Instance Types
# Include "null" to allow missing or computed values
allowed_types = ["n1-standard-1", "n1-standard-2", "n1-standard-4"]

# Get all GCE instances
allGCEInstances = plan.find_resources("google_compute_instance")

# Filter to GCE instances with violations
# Warnings will be printed for all violations since the last parameter
is true
violatingGCEInstances =
plan.filter_attribute_not_in_list(allGCEInstances,
                        "machine_type", allowed_types, true)

# Count violations
violations = length(violatingGCEInstances["messages"])

# Main rule
main = rule {
  violations is 0
}
```

Step 6: Test your Sentinel policy with the Sentinel CLI:

After creating your Terraform configuration in Step 1, creating your workspace in Step 2, running a plan in Step 3, generating mocks in step 4, and writing your Sentinel policy in Step 5, you can begin to test your policy with the Sentinel CLI.

The Sentinel CLI includes the `sentinel test` command that allows you to test Sentinel policies against test cases and mocks. Test cases are json files that indicate what results specific rules such as `main` should have and what mocks should be used by the test cases. Sentinel is particular about the organization of test cases and mocks. Assuming that the sentinel binary is in your path, you can place Sentinel policies in any directory and test them from it, but you need to create a test directory underneath that directory and then create sub-directories with the same names as the policies (without the ".sentinel" extension) underneath the test directory.

So, if you want to test the restrict-gce-machine-type.sentinel policy in a directory called "gcp-policies", you would create a "test" directory under "gcp-policies" and a "restrict-gce-machine-type" directory under "test".

While the Sentinel CLI includes an `apply` command, we will only discuss the `test` command in this section and will assume that you have created the above directories.

Before you can test the restrict-gce-machine-type.sentinel policy, you need to define test cases and make them reference copies of the tfplan mock you generated in Step 4 as well as the tfplan-functions Sentinel module. You should create pass.json and fail.json files that look like these:

pass.json:

```
{
  "modules": {
    "tfplan-functions": {
      "path":
"../../../common-functions/tfplan-functions/tfplan-functions.sentinel"
    }
  },
  "mock": {
    "tfplan/v2": "mock-tfplan-pass.sentinel"
  },
  "test": {
    "main": true
  }
}
```

fail.json:

```
{
  "modules": {
    "tfplan-functions": {
      "path":
"../../../common-functions/tfplan-functions/tfplan-functions.sentinel"
    }
  },
  "mock": {
    "tfplan/v2": "mock-tfplan-fail.sentinel"
  },
  "test": {
    "main": false
  }
}
```

Note that the fail.json test case expects the main rule to return false. In other words, the main rule should fail, but that will cause the test case to pass.

You should then copy the mock-tfplan.sentinel file you generated in Step 4 to the "gce-policies/test/restrict-gce-machine-type" directory and rename it "mock-tfplan-pass.sentinel". Also create a copy of it called "mock-tfplan-fail.sentinel".

Since our policy requires that the machine_type attribute of the google_compute_instance resource instances be in the list that includes n1-standard-1, n1-standard-2, and n1-standard-4, edit the "mock-tfplan-pass.sentinel" file so that all occurrences of "machine_type" do have one of those values. Edit the "mock-tfplan-fail.sentinel" file so that all occurrences of "machine_type" are n1-standard-8. (You will find values of "machine_type" both under the "applied" and "diff" sections of each google_compute_instance resource in the mock files.)

You can now run your test from the "gcp-policies" directory with this command:

```
sentinel test -run=restrict-gce-machine-type
```

Note that you only need to type enough of the policy name to give a unique match within the set of policies in the current directory for the `-run` argument.

Here is what you should see:

```
$sentinel test -run=restrict-gce-machine-type -verbose
PASS - restrict-gce-machine-type.sentinel
  PASS - test/restrict-gce-machine-type/fail.json
    logs:
      Resource google_compute_instance.demo[0] has attribute machine_type with value
n1-standard-8 that is not in the allowed list: ["n1-standard-1" "n1-standard-2" "n1-standard-4"]
    trace:
      FALSE - restrict-gce-machine-type.sentinel:94:1 - Rule "main"

  PASS - test/restrict-gce-machine-type/pass.json
    trace:
      TRUE - restrict-gce-machine-type.sentinel:94:1 - Rule "main"
```

If both test cases do not pass and the text is not green for both, then double-check your policy and mocks to make sure they conform to what we specified above.

Step 7: Revise Your Policy And Test Cases Until They All Pass:

A complete, successful test process for any Sentinel policy requires that it pass when run against test cases with mocks which satisfy its main rule and that it fail when run against test cases with mocks that violate its main rule.

If your policy only evaluates a single condition, you will only need one pass test case and one fail test case. But, if your policy evaluates four conditions, all of which are supposed to be true, you

should create one pass test case that satisfies all four conditions, one that fails the first condition, one that fails the second condition, one that fails the third condition, and one that fails the fourth condition. It is also good to create a test case that fails all four conditions.

If any of your test cases do not pass or if you get errors with red text, revise your policy and test cases until they do all pass.

Step 8: Deploy Your Policy to an Organization on a Terraform Server:

After all your Sentinel CLI test cases do pass, add the policy to an organization on a Terraform server and test it between the plan and apply of an actual Terraform run against the workspace you created in Step 2. Be sure to add it to a policy set that is applied to that workspace. Initially, you might want to create a policy set that only contains your new policy and make sure that it is the only policy set applied to your workspace. This will avoid having to look at the results of other policies if there are any violations of the new policy.

If you include variables in your Terraform configuration and workspace that you test the policy against, you can set them to different values to make sure that the policy passes when all the variables have passing values and that it fails when any of them have failing values. In some cases, especially if your policy tests for the presence of certain attributes, full testing of a policy on your Terraform server might require more than one workspace that use slightly different versions of your Terraform code. In any case, if you constructed adequate test cases when testing your policy with the Sentinel CLI, then you should not have any unexpected problems when testing and using your policy on your Terraform server.

About the Exercises in this Guide and Workshop

One of the main objectives of this guide and the associated Sentinel for Terraform Workshop is to make you capable of independently writing and testing new Sentinel policies. The exercises in this guide and the hands-on Instruqt tracks of the workshop therefore do not give you a simple set of steps to follow in which you could blindly copy and paste snippets of Sentinel code into policies. Instead, you have to look at documentation for various Terraform providers and underlying technologies and figure out how to complete policies that have been partially written for you. We believe you will learn more if we force you to think a bit while you complete the policies in the exercises.

However, everything you need to know is covered in this guide. If you get stuck, re-read relevant parts of the guide or the workshop slides, check out the links we have provided, see the example policies in the governance section of the terraform-guides repository, review the notes screens of the Instruqt challenge, or click the green Instruqt Check button to get a hint. As a last resort, see the solutions to the exercises in this repository.

We encourage you to do each exercise when you reach it either in this guide or in the workshop slides before reading beyond it so that the immediately preceding material is fresh in your mind.

The workshop has two hands-on Instruqt tracks that are completely self-contained and can be run from any browser:

1. Sentinel CLI Basics teaches you the basics of the Sentinel CLI
2. Sentinel for Terraform v4 guides you through the 5 exercises and an extra credit challenge.

The tracks contain all the mocks and test cases needed for use with the Sentinel CLI and partially written policies that you need to complete and then test. In exercises 2 - 5, you will need to complete two different versions of the policy. Complete the policies by removing placeholders like <resource_type> with suitable Sentinel expressions or values. Since the tracks only use the Sentinel CLI and do not actually run any Terraform code, you do not need any cloud credentials to complete them.

After completing each policy, run the `sentinel test` command against it. This should only return green text indicating that all test cases passed. If you see any red text or if any of the test cases gives a message saying that an error occurred and refers you to a line within your sentinel policy, then you need to modify your policy. If you just had a typo or failed to include a closing brace, then the change you need to make might be obvious. But it is also possible that you have an error of logic or that one of your expressions is giving `undefined`. If you click the green Check button in the challenge, you will get a hint that should help you fix your policy.

Exercise 1

In this exercise, you will complete and test a policy very similar to the two main examples we covered when explaining our methodology. It uses the tfplan/v2 import.

Challenge: Complete and test a Sentinel policy that restricts Vault authentication (auth) methods (backends) created by Terraform's Vault Provider to the following choices: Azure, Kubernetes, GitHub, and AppRole. Follow the methodology given above.

Make sure the policy passes when any of these types are specified and fails when other types are specified.

Vault is HashiCorp's secrets management solution. Please complete Exercise 1 even if you do not use Vault since doing so will improve your understanding of Sentinel.

Solution: See the restrict-vault-auth-methods.sentinel policy and the restrict-vault-auth-methods test case directory in the solutions repository.

Some Useful Sentinel Operators, Functions, and Concepts

In this section, we cover some additional Sentinel concepts. First, we refer you to some Sentinel documentation about comparison operators which can be used in functions and rules. The most common operators in addition to the set operators "in" and "contains" are the equality

operators, "==" and "is", both of which test if two expressions are equal, and the inequality operators "!=" and "is not", which test if two expressions are not equal. You can also use the mathematical operators "<", "<=", ">", and ">=" for numerical comparisons.

We also want to mention the logical operators which can be used to combine boolean expressions. These include "and", "or", "xor", and "not", the last of which can also be expressed with "!". While "or" returns true if either or both of its arguments are true, "xor" is an exclusive or which only returns true if exactly one of the two arguments is true.

It is possible to combine multiple rules in your main rule with these logical operators. For example, a main rule could require two other rules to be true with the "and" operator like this:

```
main = rule {
  ruleA and ruleB
}
```

If you had two rules of which at least one (and possibly both) should be true, your main rule would look like this:

```
main = rule {
  ruleA or ruleB
}
```

If you had two rules of which exactly one (but not none or both) should be true, your main rule would look like this:

```
main = rule {
  ruleA xor ruleB
}
```

It's also important to understand that Sentinel applies short-circuit logic to logical operators from left-to-right. This includes the "all" and "any" loops which are converted to chained "and" and "or" operators respectively. As soon as Sentinel can determine the result of a boolean expression, including those returned by rules, it stops processing the expression and returns that value. So, in the expression "(2*5 == 11) and (5 + 1 == 6)", the second part is never evaluated since the first part is false and Sentinel knows that the entire expression must be false. Likewise, in the expression "(2*5 == 10) or (5 + 1 == 7)", the second part is never evaluated since the first part is true and Sentinel knows that the entire expression must be true. In an "all" loop, Sentinel stops evaluating instances as soon as one violates the condition.

Because of Sentinel's short-circuit logic, I actually recommend that all Terraform Sentinel policies only ever use a single `main` rule that processes a boolean variable set before the rule by calling one or more functions that use all loops to iterate over all instances of specific resource types. Doing this allows your policies to print violation messages for all resource instances that violate any of your their conditions. In contrast, if you use a main rule with other

rules as shown above, some of your rules might not be invoked which will prevent printing of their violation messages.

You might also find the matches operator, which tests if a string matches a regular expression, useful. When creating regex for use with Sentinel, you might find the Golang Regex Tester website useful. Keep in mind, however, that you will need to escape any use of the "\" character in the regex expressions you actually use in your policy (but not on that website). The reason for this is that Sentinel allows certain special characters to be escaped with "\"; so, using something like "\." in your regex expression causes Sentinel to give an error "unknown escape sequence" since "\." is not one of the valid escape sequences. So, if you wanted to use "(.+)\.acme\.com$" to match domains like "www.acme.com", you would use that on the Golang Regex Tester website but would use "(.+)\\.acme\\.com$" inside your policy.

Use of the built-in Sentinel length function is also quite common to ensure that an attribute was actually included in all occurrences of a specific resource in all Terraform code. Note, however, that it usually needs to be combined with the else operator as in these equivalent examples:

```
1. (length(tags) else 0) > 0
2. (length(tags) > 0) else false
```

The important thing to understand in these examples is that if the resource does not have a `tags` attribute, then `length(tags)` will evaluate to `undefined`. In the first example, the expression "`length(tags) else 0`" gives the length of `tags` if it exists but gives 0 if it is `undefined`. In the first case, the rule then requires that the length of the tags attribute be greater than 0. In the second case it requires that 0 > 0 which will give `false`.

In the second example, if the resource does not have the tags attribute, then `length(tags)` will again evaluate to `undefined` and so will `length(tags) > 0`, but the inclusion of "`else false`" will convert that to `false`.

The two examples give the exact same results for any resource, returning true when it has tags and false when it does not. You can use whichever syntax you find more appealing.

Other built-in Sentinel functions are listed here.

You might also find functions in the Sentinel strings import and other standard imports useful. The strings import has operations to join and split strings, convert their case, test if they have a specific prefix or suffix, and trim them.

Exercise 2

In this exercise, you will complete two versions of a policy that restricts the creation of AWS access keys so that the associated secret keys returned by Terraform are encrypted by PGP keys and therefore cannot be decrypted by anyone looking at Terraform state files unless they have the PGP private key that matches the PGP public key that was specified when creating the access keys.

Challenge: Complete and test two versions of a Sentinel policy that requires all AWS IAM access keys created by Terraform to include a PGP key.

Make sure that the policy passes when the Terraform code does provide a PGP key and fails when it does not.

Note: The PGP key used for creating an IAM access key can be given in the format "keybase:<user>" or by providing a base-64 encoded PGP public key. For this exercise, you can assume that only the first option would ever be used.

Sentinel's purpose is not to ensure that Terraform code can actually be applied. If the Terraform code tries to reference an invalid AWS user or PGP key, the apply will fail, but the Sentinel policy would still be doing what it is supposed to do: prevent creation of AWS keys that don't even include a PGP key.

Solution: See the require-access-keys-use-pgp-a.sentinel and require-access-keys-use-pgp-b.sentinel policies and their associated test case directories in the solutions repository.

Printing in Sentinel Policies

One of the most useful Sentinel functions is the built-in print function. You can call it in your own rules and functions. This section documents how to use the print function correctly inside rules so that you do not alter the results of those rules. However, in general, it is preferable to make rules call functions and do your prints inside the latter. Using functions avoids Sentinel's short-circuit logic and allows you to print violation messages for all resource instances that violate conditions.

Including the print function in all of your functions and rules is a best practice since it makes it easier for people looking at a Sentinel log to understand why a policy failed against their run. We encourage you to do this in your solutions to the exercises even if you are confident that your policies will execute correctly. *Printing is not just for debugging!*

The return value of the print function is always `true`; this allows you to use it in a rule as long as you combine it with other boolean expressions in the rule using one of Sentinel's logical operators, ensuring that your rule still returns the same boolean value that it would have returned without the print function. (If your rule already includes multiple conditions, you might need to add parentheses around them.)

However, as previously mentioned, we recommend that you always use a single `main` rule in your Terraform Sentinel policies and have it evaluate a boolean variable generated by calling one or more Sentinel functions. If you do this, you will use the print function in your functions and will not need to include it in any rules.

When you include the print function in your functions and rules, the printed text will show up in the Sentinel log above Sentinel's default output for that policy. There are two reasons why you might not see a print statement:

1. Whatever you tried to print had the `undefined` value inside it. In this case, you should try printing the Sentinel structure above the item you previously tried to print. For example, you could try printing `key.change.after` instead of `key.change.after.pgp_key` if the latter did not print anything.
2. It is also possible that Sentinel's short-circuit logic that we discussed above caused Sentinel to stop processing your rule before it got to the print function.

Evaluating Data Sources in Sentinel Policies

Restricting the attributes of data sources in Sentinel policies is very similar to restricting the attributes of resources.

However, it is important to understand that Terraform data sources that do not reference any computed values are actually evaluated by Terraform during the refresh operation done by the plan and that the results are stored in the state of the workspace. (If a data source does reference computed values, then it will be evaluated during the apply; but Sentinel can then not evaluate it at all.) While the tfplan/v2 import does combine current state with planned changes, the values of attributes of data sources that have just been evaluated do not show up in it.

So, your Sentinel policy will need to use the tfstate/v2 import or the tfstate-functions Sentinel module to evaluate data sources.

You can look at this example from the terraform-guides repository as well as the tfstate/v2 import documentation.

While referencing data sources in Sentinel policies is a bit trickier than referencing resources, once you write your first policy that uses them, you won't have any problems writing others.

Exercise 3

In this exercise, you will complete two versions of a policy that restricts any data source that retrieves an Amazon Certificate Manager (ACM) certificate to have a subdomain within a specific domain. A customer might want a policy like this to make sure that all ACM certificates are for one or more specified domains so that they can ensure that they are monitoring all of their websites.

Challenge: Complete and test two versions of a Sentinel policy that requires all Amazon Certificate Manager (ACM) certificates returned by Terraform data sources to be subdomains of some higher level domain. Use Sentinel's print() function to print domains for your certificates.

For instance, here at HashiCorp, we might require that all certificates have domains that are subdomains of "hashicorp.com". This means the domain would have to end in ".hashicorp.com".

Make sure that the policy passes when all certificates have domains that are subdomains of the domain "hashidemos.io" but fails if any certificate has a domain that is not a subdomain of "hashidemos.io". For instance, here at HashiCorp, our policy should pass for domains such as "docs.hashidemos.io" but fail for domains such as "docs.hashydemos.io".

While the policy could be written with the Sentinel `matches` operator or the Sentinel `strings` import, you should use the `matches` operator in this exercise. A valid regex to use would be something like "(.+)\\.hashidemos\\.io$". (See the earlier explanation of why we have to escape the "\" in regular expressions.)

Solution: See the restrict-acm-certificate-domains-a.sentinel and restrict-acm-certificate-domains-b.sentinel policies and their associated test case directories in the solutions repository.

Dealing with Lists, Maps, and Blocks

Many Terraform resources include lists, maps, and blocks in addition to their top-level attributes. A list in a resource contains multiple primitive data types such as strings or numbers. In Terraform code lists are indicated with rectangular brackets: []. A map specifies one or more key/value pairs in a resource and is indicated with curly braces: {}. A block is a named construct in a resource that has its own section within the documentation for that resource. Each block is indicated with curly braces just like a map, but since some blocks can be repeated, they are represented inside Terraform plans as lists of maps. Sentinel also sees blocks as lists of maps, using curly braces for each block within a list indicated with rectangular brackets. Because maps and blocks are indicated with the same curly braces it is not always clear what a structure within a Terraform resource actually is. In this section, we provide techniques that will help you understand how these structures are being used by your resources and how you can restrict their attributes with Sentinel policies.

There are two ways to see how the lists, maps, and blocks used by your resources are organized:

1. Look at plan logs for the Terraform code you have written to test your policies.
2. Use the Sentinel print function to print out the entire resource.

You could also use both methods together. We're going to provide a few examples and illustrate them in some detail because we have found it very helpful to fully understand how Terraform and Sentinel format plans and policy checks.

Terraform Plan Format:

Here is a Terraform log that shows part of a plan for a single Azure VM:

```
  + module.windowsserver.azurerm_virtual_machine.vm-windows
```

```
    id:                                             <computed>
    location:                                       "eastus"
    name:                                           "demohost0"
    resource_group_name:                            "rogerberlind-win-rc"
    storage_image_reference.#:                      "1"
    storage_image_reference.3904372903.id:          ""
    storage_image_reference.3904372903.offer:       "WindowsServer"
    storage_image_reference.3904372903.publisher:   "MicrosoftWindowsServer"
    storage_image_reference.3904372903.sku:         "2016-Datacenter"
    storage_image_reference.3904372903.version:     "latest"
    storage_os_disk.#:                              "1"
    storage_os_disk.0.caching:                      "ReadWrite"
    storage_os_disk.0.create_option:                "FromImage"
    storage_os_disk.0.disk_size_gb:                 <computed>
    storage_os_disk.0.managed_disk_id:              <computed>
    storage_os_disk.0.managed_disk_type:            "Premium_LRS"
    storage_os_disk.0.name:                         "osdisk-demohost-0"
    tags.%:                                         "1"
    tags.source:                                    "terraform"
    vm_size:                                        "Standard_DS1_V2"
```

This resource shows two blocks ("storage_image_reference" and "storage_os_disk") and one map ("tags"). The size of the blocks is given with the "<block_name>.#" attributes while the size of the "tags" map is given by the "tags.%" attribute. (All three have the value "1" in this case.) Since a block can be repeated, each instance of a block in the plan will also have an index or identifier. Some blocks such as "storage_os_disk" are indexed starting with 0 while others such as "storage_image_reference" have identifiers like "3904372903". Fortunately, this difference won't affect how we write Sentinel policies to examine these blocks.

Each instance of a block is structured like a map with each key/value pair including the key as part of the attribute name and the value as the value of the attribute. So, the single instance of the "storage_image_reference" block has a key called "offer" with value "WindowsServer". This is very similar to what we see for the resource's "tags" map, which has a single key "source" with value "terraform". However, maps do not have indices or identifiers the way blocks do.

Finally, we want to mention that the size of a list is always indicated with the "<list>.#" attribute and that lists are always indexed starting with 0. Here is what a list in a plan for a single ACM Certificate (aws_acm_certificate) might look like along with a "tags" map that has two attributes.

```
+ aws_acm_certificate.new_cert
    domain_name:                  "roger.hashidemos.io"
    subject_alternative_names.#:  "3"
    subject_alternative_names.0:  "roger1.hashidemos.io"
    subject_alternative_names.1:  "roger2.hashidemos.io"
    subject_alternative_names.2:  "roger3.hashidemos.io"
    tags.%:                       "2"
    tags.owner:                   "roger"
```

```
      tags.ttl:                     "24"
```

Sentinel Print Output Format:

If you use the Sentinel print function, look at the output of your Sentinel policy applied against a workspace that uses Terraform code that creates the resource or against mocks generated from a run against that workspace. Here is some Sentinel output that shows the equivalent information of the above Terraform plan for the same Azure VM. Note that we reformatted the original output, which was all on a single line, to make it more readable in this guide:

```
{
  "id": "74D93920-ED26-11E3-AC10-0800200C9A66",
  "location": "eastus",
  "name": "demohost0",
  "resource_group_name": "rogerberlind-win-rc",
  "storage_image_reference": [
    {
      "id": "",
      "offer": "WindowsServer",
      "publisher": "MicrosoftWindowsServer",
      "sku": "2016-Datacenter",
      "version": "latest"
    }
  ],
  "storage_os_disk": [
    {
      "caching": "ReadWrite",
      "create_option": "FromImage",
      "disk_size_gb": "74D93920-ED26-11E3-AC10-0800200C9A66",
      "managed_disk_id": "74D93920-ED26-11E3-AC10-0800200C9A66",
      "managed_disk_type": "Premium_LRS",
      "name": "osdisk-demohost-0"
    }
  ],
  "tags": {
    "source": "terraform"
  },
  "vm_size": "Standard_DS1_V2"
}
```

Here is what the Sentinel output looks like for the above ACM certificate after reformatting:

```
{
 "arn": "74D93920-ED26-11E3-AC10-0800200C9A66",
 "domain_name": "roger.hashidemos.io",
 "domain_validation_options": "74D93920-ED26-11E3-AC10-0800200C9A66",
```

```
 "id": "74D93920-ED26-11E3-AC10-0800200C9A66",
 "subject_alternative_names": [
  "roger1.hashidemos.io",
  "roger2.hashidemos.io",
  "roger3.hashidemos.io"
 ],
 "tags": {
  "owner": "roger",
  "ttl": "24"
 },
 "validation_emails": "74D93920-ED26-11E3-AC10-0800200C9A66",
 "validation_method": "DNS"
}
```

Differences Between Terraform Plans and Sentinel Print Output:

There are some differences between the Terraform plans and the Sentinel outputs:

1. The Sentinel output is formatted in JSON.
2. Lists use brackets, maps use braces, and blocks use brackets and braces with the brackets surrounding all instances of a particular kind of block and the braces surrounding each instance of the block.
3. The sizes of lists, maps, and blocks are not included.
4. The indices and identifiers of blocks and lists are not included.

The fact that the indices and identifiers of blocks are not included is why we don't need to worry about whether a block is indexed starting with 0 or uses other identifiers; in either case, you can just use a "for" loop to iterate over all instances of the block and apply your desired conditions to them. However, we will see below that a specific syntax must be used when doing this.

Whenever there are differences between a Terraform plan log and the output of the Sentinel print function for lists, maps, or blocks within the same resource, rely on the print function output when writing your policy because that is what your policy needs to match. However, plan logs do have nicer formatting and can usually give you a good sense for how to reference these structures.

Referring to Lists, Maps, and Blocks in Sentinel Policies:

You can iterate across all members of a list with a `for` loop inside a function to examine or process all members of a list.

Here is an example of a function which prints all members of the subject_alternative_names list for all instances of the aws_acm_certificate resource. Note that we often only use one variable after `as` when iterating over lists. The first `for` loop in this example uses two variables because it is iterating over a map which has keys and values.

```
print_acm_sans = func(acm_certs) {
  for acm_certs as address, cert {
    for cert.subject_alternative_names as san
      print( "SAN:", san)
    }
  }
}
```

You can iterate over keys and values of maps in a similar fashion. Here is an example that prints out the tags for all instances of the same aws_acm_certificate resource, giving the keys and values:

```
print_acm_tags = func(acm_certs) {
  for acm_certs as address, cert {
    for cert.tags as k, v {
        print( "key:", k, ", value:", v )
    }
  }
}
```

For maps, we provide two variables after `as`, one for the keys of the map and one for the values of the map.

It is also possible to refer to the values of specific keys in a map in two different ways: using `<map>.<key>` or `<map>["<key>"]` in which `<map>` is the name of the map and `<key>` is a specific key. If we wanted to make sure that the `ttl` key of the `tags` map of each aws_acm_certificate resource was set to "24", we could use this function (in which we have shown the second way of referencing the key in a comment):

```
print_acm_tags = func(acm_certs) {
  for acm_certs as address, cert {
    if cert.tags.ttl is "24" {
    #if cert.tags["ttl"] is "24" {
      print( "ttl is allowed" )
      return true
    } else {
      print("ttl is not allowed")
      return false
    }
  }
}
```

While keys of Sentinel maps can be strings, numerics, or booleans, they will mostly be strings, and you will need to use double quotes around the name of the key as we did above. Values in maps can be of any type, so you might or might not need quotes for them. We recommend using double quotes around all values including numeric ones and only removing them if you

get a Sentinel error like "comparison requires both operands to be the same type, got string and int".

We can now illustrate how you can refer to blocks and even nested blocks in your Sentinel rules. The techniques are very similar to what we did above for lists and maps since blocks are lists of maps and a block with nested blocks is a list of maps containing lists of maps.

Let's use the azurerm_virtual_machine resource for which we showed a Terraform plan log and Sentinel print function output above. Recall that it had two blocks ("storage_image_reference" and "storage_os_disk") each of which had a single instance of a map. Here is some Sentinel code which requires the publisher to be "RedHat":

```
if r.change.after.storage_image_reference[0].publisher is not "RedHat"
{
  print("VM", address, "has publisher",
  r.change.after.storage_image_reference[0].publisher,
  "that is not allowed")
}
```

Because a block always starts with a list and we expect the storage_image_reference block to only have one instance, we referred to "`storage_image_reference[0]`". This uses what we learned about lists. Since each instance of the storage_image_reference block is a map and we are interested in the publisher key of that map, we appended "`.publisher`" to that. This uses what we learned about maps. We then require the value to be "RedHat".

If you examine the Terraform plan that we showed for an Azure VM above, you might think that you could also use "`storage_image_reference.3904372903.publisher`" in the rule above. However, this will give a Sentinel error because the Sentinel output for the same Azure VM resource did not include the 3904372903 identifier. This is one of the cases we had in mind when we suggested above that you should rely on the Sentinel print function output instead of on the Terraform plan when they differ. The alternative version of the rule won't work because Sentinel doesn't know anything about the 3904372903 identifier.

Sometimes, an attribute of a resource might have different types depending on whether it was included in the Terraform code or not. In particular, attributes that are computed in Terraform 0.11 are represented by a placeholder string containing a UUID used for all computed values. For example, the versioning attribute of the aws_s3_bucket resource will be a string if it is not included in the code but will be a block if it is included. In cases like this, if you try to refer to a nested attribute of the versioning block with something like `r.applied.versioning[0].enabled` (using the old v1 tfplan import), you will get a hard Sentinel error saying "only a list or map can be indexed, got string" whenever the versioning attribute is not included in the Terraform code.

Fortunately, we can handle this scenario with the Sentinel standard import types. In a function that needs to check that the boolean `r.applied.versioning[0].enabled`, we first check that the type of `r.applied.versioning` is a list like this:

```
for buckets as address, r {
  if types.type_of(r.applied.versioning) else "" is "list" and
     r.applied.versioning[0].enabled is false {
     print("S3 bucket", address,
           "does not have versioning enabled")
     validated = false
  }
}
```

Sentinel's short-circuit logic ensures that `r.applied.versioning[0].enabled` will only be evaluated if the type of `r.applied.versioning` actually is a list. This rule will work whether the versioning attribute is included in the Terraform code or not. Note that you must import the types import at the top of your policy with `import "types"` before you can use it.

Exercise 4

In this exercise, you will complete two version of a Sentinel policy that restricts all Google virtual machine instances (google compute instances) to use a specific public image. You will need to refer to a nested block of a particular block of the google_compute_instance resource to evaluate the image.

Challenge: Complete and test two versions of a Sentinel policy that requires all Google compute instances provisioned by Terraform to use the public image "debian-cloud/debian-9".

Make sure the policy passes when the image specified for the compute instance is "debian-cloud/debian-9" and fails when other images are specified. You could use "centos-cloud/centos-7" to test failure of the policy.

Solution: See the restrict-gcp-instance-image-a.sentinel and restrict-gcp-instance-image-b.sentinel policies, and their associated test case directories in the solutions repository.

Using the Sentinel tfconfig Import

So far, we've discussed several examples of using the tfplan import with Terraform and one example of using the tfstate import. This section gives an example of using the tfconfig/v2 import, which can inspect the Terraform configuration files that are run within a Terraform workspace. Keep in mind that the tfconfig/v2 import examines the actual code expressions assigned to various attributes rather than the values that are ultimately interpolated.

In our example, we will show how to restrict which provisioners can be used with null resources. Specifically, we will show a Sentinel policy that prevents them from executing remote-exec provisioners. Note that the tfconfig import is the only Sentinel import that can evaluate provisioners being attached to resources. It is also useful for restricting providers, modules, and other types of objects within Terraform configurations.

Here is a snippet of Terraform code that creates a null resource and uses a remote-exec provisioner to run a script on a remote host:

```
resource "null_resource" "post-install" {
  provisioner "remote-exec" {
    script = "${path.root}/scripts/postinstall.sh"
    connection {
      host = "${var.host}"
      type = "ssh"
      agent = false
      user = "ec2-user"
      private_key = "${var.private_key_data}"
    }
  }
}
```

Note that the provisioner is a block under the null resource. Any resource can have zero, one, or multiple provisioner blocks.

A complete Sentinel policy that prevents null resources from using remote-exec provisioners is in the terraform-guides repository. Here, we just show the portion that restricts the type of provisioners allowed:

```
# Get remote-exec provisioners
remoteExecProvisioners =
config.find_provisioners_by_type("remote-exec")

# Filter to provisioners with violations
# Warnings will not be printed for violations since the last parameter
is false
violatingProvisioners = filter remoteExecProvisioners as address, p {
  strings.has_prefix(p.resource_address, "null_resource.") and
  print("Provisioner", address, "of type", p.type, "is not allowed.")
}
```

If you look at the full policy, you'll notice that the overall pattern of this policy is very similar to the ones we wrote to restrict resources. We use one function that retrieves remote-exec provisioners and a second function that validates whether the resource that uses them is a null_resource. The main rule checks that the number of violations is 0.

Note that if you want to refer to attributes of resources evaluated with the tfconfig/v2 import, you need to use the `constant_value` or `references` fields. The first will have data when an expression was set to a constant value like a string or number. The second will be used when the expression has references to variables or to attributes of other resources or data sources. Note that when static values are mixed with references, the static values will not be visible to the tfconfig/v2 import.

For an example of a policy which uses these fields, see the prohibited-local-exec-commands.sentinel policy.

Exercise 5

In this exercise, you will complete two versions of a Sentinel policy that requires all Azure modules to come from the Private Module Registry (PMR) of your organization on your Terraform server.

Challenge: Complete and test two versions of a Sentinel policy that requires all Azure modules directly under the root module of any Terraform configuration to come from the Private Module Registry of your organization on your Terraform server. Then use the Sentinel CLI to test your policy using the `test` command.

By restricting the modules that are used by the root module to come from the PMR, we can effectively require that all modules other than root modules come from the PMR since the team managing the modules in the PMR could ensure that they do not use external modules. We could also prevent any resources from being created in the root module by adding a rule that requires that `length(tfconfig.resources) == 0`.

For this exercise, you will complete a Sentinel policy and then use the Sentinel CLI to test it against the mocked data we have provided.

The source for a module from the public Terraform Module Registry is <namespace>/<module name>/<provider>. So, the Azure network module in the public registry has source "Azure/network/azurerm".

The source for a module from a private module registry is <Terraform hostname>/<Terraform organization>/<module name>/<provider>. So, the same Azure network module in a private registry in the Cloud-Ops organization on the SaaS Terraform server running at https://app.terraform.io would have source "app.terraform.io/Cloud-Ops/network/azurerm".

Make sure that your Sentinel policy passes when all modules come from your PMR but fails if any module comes from a different source.

Solution: See the require-modules-from-pmr-a.sentinel and require-modules-from-pmr-b.sentinel policies and their corresponding test case directories in the solutions repository.

Extra Credit

In this extra credit challenge, you will use the tfrun import to complete and test a policy that prevents the use of auto-apply in production workspaces.

Challenge: Complete and test a Sentinel policy that prevents auto-apply from being set on production workspaces. Then use the Sentinel CLI to test your policy using both the `apply` and `test` commands.

The policy checks metadata for each workspace including its name and auto-apply setting.

Solution: See the prevent-auto-apply-in-production.sentinel policy and its corresponding test case directory in the solutions repository.

Conclusion

In this guide, we have provided an in-depth survey of writing and testing Sentinel policies in Terraform. We discussed different types of Sentinel policies that use Terraform's tfplan, tfconfig, tfstate, and tfrun imports. We described three methods of testing Sentinel policies and discussed their pros and cons. We laid out a basic eight-step methodology for writing and testing new Sentinel policies that restrict attributes of resources. This methodology includes creating Terraform configurations and workspaces, running plans and generating mocks against them, writing Sentinel policies, and testing of these policies with the Sentinel CLI until the policies behave correctly.

We also reviewed some useful Sentinel operators, functions, and concepts. These included the comparison, logical, and matches operators, the length function, and the strings and types imports, all of which can be very useful in Sentinel policies. We also gave some pointers on using Sentinel's print function in your policies, both for debugging any problems with them, but also to make the outputs of your policies clearer to users whose workspaces violate them.

We expanded our coverage of Sentinel to show you how to restrict the results of Terraform data sources and deal with lists, maps, and blocks within resources. We also showed an example of using the tfconfig import. Finally, we showed an additional example of using the Sentinel CLI with a tfconfig mock so that you can automate your testing of your Sentinel policies.