HashiCorp

Terraform

Writing and Testing Sentinel Policies for Terraform

Contents

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 Simulator to test your policies. It also discusses the differences between policies meant for use with Terraform 0.11.x and 0.12.x. Finally, it includes examples and exercises to practice writing Sentinel policies and test your understanding of them.

Types of Sentinel Policies for Terraform

There are essentially three types of Sentinel policies for Terraform which correspond to these three Sentinel imports: tfplan, tfconfig, and tfstate. 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. We expect additional Sentinel imports to be added to Terraform in the near future to allow Sentinel policies to inspect workspace metadata attributes and cost estimates.

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 Terrafrom 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 Simulator 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 Simulator since TFE has to run a plan against your workspace before it even invokes any Sentinel policies.
3. Unless you use TFE 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 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

In order to use the methodology described in this guide, you will 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 Simulator, 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 Simulator.

Downloading and Installing the Sentinel Simulator

Download the Sentinel Simulator for your OS from the Sentinel Downloads page. We used version 0.10.2 which was the most current version at the time we wrote this guide. 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 Simulator 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 Simulator
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 a Sentinel policy to restrict the size of Google compute instances by borrowing code from the restrict-aws-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.

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-google-compute-instance-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 Use 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 "None" as the source 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 0.11.x or 0.12.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 three 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 three files: mock-tfconfig.sentinel, mock-tfplan.sentinel, and mock-tfstate.sentinel. We will primarily use the mock-tfplan.sentinel file.

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 Simulator 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 restrict-ec2-instance-type policy. Like many other

Sentinel policies in the terraform-guides repository, this policy uses several common functions including find_resources_from_plan and validate_attribute_in_list. These are parameterized functions that can be re-used in many policies. You will therefore not have to change very much code when creating a new policy from an existing one. Additionally, the policy and functions follow the recommendations given in the Sentinel Defining Policies documentation.

Each Sentinel policy intended to restrict attributes of resources or data sources will use the Sentinel tfplan and strings imports by including these statements at the top:

```
import "tfplan"
import "strings"
```

The find_resources_from_plan function retrieves all instances of a specified resource from all modules within the diff 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_from_plan function with minor formatting changes:

```
# Find all resources of a specific type from all modules
# using the tfplan import
find_resources_from_plan = func(type) {

  resources = {}

  # Iterate over all modules in the tfplan import
  for tfplan.module_paths as path {
    # Iterate over the named resources of desired type in the module
    for tfplan.module(path).resources[type] else {} as name, instances {
      # Iterate over resource instances
      for instances as index, r {

        # Get the address of the instance
        if length(path) == 0 {
          # root module
          address = type + "." + name + "[" + string(index) + "]"
        } else {
          # non-root module
          address = "module." + strings.join(path, ".module.") + "." +
                    type + "." + name + "[" + string(index) + "]"
        }

        # Add the instance to resources map, setting the key to the address
        resources[address] = r
      }
    }
  }

  return resources
}
```

This function accepts a single parameter called `type` which specifies the type of resource we want to find. We will see later that this is passed to the function with a string like "aws_instance". We start by creating an empty map called `resources`. We then loop through all modules in the current plan with a `for` loop, referring to each module found as path. We then loop through the named resources of each module and the instances of each named resource so that we can add all instances of the resource of the given type from all modules to the resources map. Doing the three loops in this function reduces the number of loops needed in validation functions.

Before adding the instances, however, we compute the full `address` of each instance, using the same address format used in plan and apply logs. Doing this allows our policies to print violation messages that fully identify the resources that violated them. Note that we treat resource instances in the root module slightly differently from resources in nested modules. After computing the address of each instance, we add it to the resources map using the address as the key.

Note the important inclusion of the else operator in "`else {}`"; this makes sure that we do not add the Sentinel value `undefined` to the resources map if the current module does not have any resources of the specified type. Doing that would make the entire map `undefined` and break the policy that calls the function.

At the end of the function we return the map of all resource instances of the specified type.
The restrict-ec2-instance-type policy also includes the validate_attribute_in_list function which validates that a specified top-level attribute of all instances of a specified resource is in a given list. Here is the code for this function:

```
# Validate that all instances of a specified resource type being modified
# have a specified top-level attribute in a given list

validate_attribute_in_list = func(type, attribute, allowed_values) {

  validated = true

  # Get all resource instances of the specified type
  resource_instances = find_resources_from_plan(type)

  # Loop through the resource instances
  for resource_instances as address, r {

    # Skip resource instances that are being destroyed
    # to avoid unnecessary policy violations
    if r.destroy and not r.requires_new {
      print("Skipping resource", address, "that is being destroyed.")
      continue
    }

    # Determine if the attribute is computed
    if r.diff[attribute].computed else false is true {
      print("Resource", address, "has attribute", attribute,
            "that is computed.")
      # If you want computed values to cause the policy to fail,
      # uncomment the next line.
      # validated = false
    } else {
      # Validate that each instance has allowed value
      if (r.applied[attribute] else "") not in allowed_values {
        print("Resource", address, "has attribute", attribute,
              "with value", r.applied[attribute] else "",
              "that is not in the allowed list:", allowed_values)
        validated = false
      }
    }
  }

  return validated
}
```

This function has three arguments: `type, attribute,` and `allowed_values`. The first is the type of the resource, the second is the top-level attribute that is being validated, and the third is the list of allowed values for that attribute.

The function defines a boolean variable called `validated` which is what the function will return. It is initially set to `true`, but will be set to `false` if any instance of the specified resource has a value of the specified attribute that is not in the list. Note, however, that it iterates over all instances even if it does find a violation. This allows the function to print violation messages for all instances that have violations.

The function then calls the `find_resources_from_plan function`, passing the type argument to it, and loops over all the instances of that type.

We then do several conditional checks:

1. We first check if the resource is being destroyed. We skip these resource instances since enforcing policies against them is usually not useful.
2. We then check if the attribute is computed, which indicates that the value cannot be determined until the apply is done. We need to do this because it is not possible to check computed values in Sentinel policies which are run between the plan and the apply of a Terraform run. By default, we do not set validated to false for computed values, but users who adopt a strict attitude with regard to policies can uncomment a line in the computed check that would do that.
3. Finally, if the value of the attribute was not computed, we check whether it is in the `allowed_values` list. If it is not, we print a violation message, but we do keep iterating across all resource instances.

In the third check, we use one of Sentinel's set operators, "`in`", to check if the attribute of each instance is in the `allowed_values` list. So, we have `(r.applied[attribute] else "") not in allowed_values`. The other set operator is "`contains`" which checks if a list contains a specific value. We also use the "`not`" and "`else`" operators.

We mentioned in the introduction that Terraform checks Sentinel policies between the plan and apply steps of runs. So, you might find the `applied` syntax confusing. Sentinel evaluates the results that the apply step would give using predicted representations of each resource's state. In other words, "`r.applied[attribute]`" gives us the value of the specified attribute for the current resource instance that would exist if the apply were allowed to run.

Finally, the function returns the value of the `validated` boolean.

We want to emphasize that you do not need to modify any of the code in the restrict-ec2-

instance-type policy from the beginning to the end of the validate_attribute_in_list function when you create a new policy from it. You will only make changes after the imports and the first two functions.

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

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

Note that a list in Sentinel must include a comma after the last item if the closing bracket is on a new line. You could also leave out the last comma and put the closing bracket on the line with the last item.

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

```
# Main rule that calls the validation function and evaluates results
main = rule {
  validate_attribute_in_list("aws_instance", "instance_type", allowed_
types)
}
```

Note that the rule calls the validate_attribute_in_list function, passing "`aws_instance`" as the type, "`instance_type`" as the attribute, and `allowed_types` as the allowed_values list.

Sentinel rules must return a boolean value, `true` or `false`. Some people prefer to define a separate rule for each condition and then combine them with a compound boolean expression in the main rule. However, we prefer to only use a main rule to reduce the amount of Sentinel output generated when defining extra rules.

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 mentioned in the main rule. 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 `machine_type` is the right attribute to restrict on the `google_compute_instance` resource, we can modify our main rule to look like this:

```
# Main rule that calls the validation function and evaluates results
main = rule {
  validate_attribute_in_list("google_compute_instance", "machine_type",
allowed_types)
}
```

Note that we have changed `aws_instance` to `google_compute_instance` and changed `instance_type` to `machine_type`.

That completes the modifications to our policy, which should be called "restrict-gce-machine-type.sentinel" and should look like the second-generation restrict-gce-machine-type.sentinel policy in the terraform-guides repository except that we have included the n1-standard-1 machine_type as valid.

We should also note that it is common to combine other rules in a policy's main rule by using Sentinel's logical operators such as "and", "or", "xor", and "not". The most common choice is to use "and" to require that all the other rules in the policy are true.

Step 6: Test your Sentinel policy with the Sentinel Simulator

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 Simulator.

The Sentinel Simulator 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 Simulator 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. You should create pass.json and fail.json files that look like these:

pass.json:

```
{
  "mock": {
    "tfplan": "mock-tfplan-pass.sentinel"
  },
  "test": {
    "main": true
  }
}
```

fail.json:

```
{
  "mock": {
    "tfplan": "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-2" "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 Simulator 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 Simulator, then you should not have any unexpected problems when testing and using your policy on your Terraform server.

About the Exercises in this Guide

One of the main objectives of this guide is to make you capable of independently writing and testing new Sentinel policies. The exercises in this guide therefore do not give you a simple set of steps to follow in which you could blindly copy and paste Terraform code and snippets of Sentinel policies into files. We believe you will learn more if you write your own code and policies.

However, everything you need to know is covered in this guide. If you get stuck, re-read relevant parts of the guide, check out the links we have provided, see the example policies in the governance section of the terraform-guides repository, or as a last resort, see the solutions to the exercises in this repository.

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

We have provided mocks that can be used for each exercise so that readers can do the exercises using the Sentinel Simulator without any cloud credentials and without deploying external technologies like Vault.

Exercise 1

In this exercise, you will create a policy very similar to the two main examples we covered when explaining our methodology.

Challenge: Create 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. You can easily download and install a Vault server if you don't already have access to one. You need a running Vault server and a valid token for it in order to run a plan against your Terraform code that uses the Vault provider so you can generate mocks against that plan. If you do not have access to a Vault server, you can use this mock with the Sentinel Simulator.

You do not need to configure the auth methods in your Terraform code or create any roles for them; just provision the auth methods themselves with Terraform. You can write Terraform code that just provisions one method at a time, but provisioning more than one at a time is a good way to test your policy's ability to handle multiple auth methods in a single workspace.

If you're not using a Vault root token, make sure that your Vault token or user has permissions to create auth methods and that you only use paths that you are allowed to use. However, you do not need to validate the path in your Sentinel policy: Terraform will take care of that if you run an apply against your workspace.

We recommend that you set the VAULT_ADDR and VAULT_TOKEN environment variables so that you do not need to include a "vault" provider configuration block in your Terraform code. Using environment variables is also more secure than embedding them in your code or in tfvars files. You'll need to set the environment variables on the Variables tab of the Terraform workspace you use to generate your mocks even if using the Terraform CLI since environment variables set in your shell for the CLI are not passed over to Terraform where the run will be executed. Please be sure to mark your VAULT_TOKEN environment variable as sensitive so that no one else can see it.

If your test 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 so, re-read the sections above.

Solution: See the vault-auth-methods directory, the restrict-vault-auth-methods.sentinel policy, and the restrict-vault-auth-methods test cases 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 quite common to combine multiple rules in your main rule with these logical operators. Most often, you will see a main rule that uses "and" like this:

```
main = rule {
  (ruleA and ruleB) else true
}
```

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) else true
}
```

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) else true
}
```

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.

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(r.applied.tags) else 0) > 0
2.  (length(r.applied.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`(`r.applied.tags`) will evaluate to `undefined`. In the first example, the expression "`length(r.applied.tags) else 0`" gives the length of `r.applied.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(r.applied.tags)` will again evaluate to `undefined` and so will `length(r.applied.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 create 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: Create and test a Sentinel policy that requires all AWS IAM access keys created by Terraform Enterprise 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.

You will need an AWS account to actually run a plan against your Terraform code and generate mocks. If you do not have one, you can use this mock with the Sentinel Simulator.

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.

The Terraform code you create does not have to reference an existing AWS user within your AWS account or a valid PGP key. These would only be needed if doing an actual apply to create the access key.

Sentinel's purpose is not to ensure that Terraform code can actually be applied. If the 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.

We recommend that you set the AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_DEFAULT_REGION environment variables so that you do not need to include an "aws" provider configuration block in your Terraform code. Using environment variables is also more secure than embedding them in your code or in tfvars files. You'll need to set the environment variables on the Variables tab of your TFE workspace even if using the Terraform CLI since environment variables set in your shell for the CLI are not passed over to TFE where the run will be executed. Please be sure to mark your AWS keys as sensitive so that no one else can see them.

Hint: There are multiple ways to implement the main condition that will check for the existence of a PGP key, but the basic framework of the policy will be similar to the two examples and the one you created in exercise 1. Instead of using the validate_attribute_in_list function, create a similar function that validates that the access key does have an associated PGP key.

Solution: See the accesskey-with-pgp and accesskey-without-pgp directories, the require-access-keys-use-pgp.sentinel policy, and the the require-access-keys-use-pgp test cases directory in the solutions repository. We give five ways of formulating the required condition.

Using Print 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; so, when you use it in a rule, you need to 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.)

As an example, if you have a policy that validates whether all S3 buckets have their ACL set to "private", are using KMS encryption and you have already set two boolean variables, "is_private" and "is_kms", and you want to print the values of those booleans in your main rule, you could do it like this:

```
main = rule {
  print("is_private:", is_private, "is_kms:", is_kms) and
  (is_private and is_kms)
}
```

By using the "and" operator after the print function, we ensure that the main rule will still return the value of `is_private and is_kms` since the expression `true and X` always has the same boolean value as `X`.

When you include the print function in your functions and rules, the printed text will show up in the Sentinel log above the rule analysis of 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 r.applied instead of r.applied.acl 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.

When you do call the print function inside rules, you can place it before or after the other conditions, with the choice depending on when you want to print. If you want to print every time the rule is evaluated, place the print function before the other conditions and add the "and" operator after it as was shown above. If you only want to print when the other conditions are false, place the print function after them and add "or not" before it. We include "not" so that the final boolean expression is still false when the other conditions are false.

We already saw the first approach above. Here is what the second approach looks like:

```
main = rule {
   (is_private and is_kms) or not
  print("is_private:", is_private, "is_kms:", is_kms)
}
```

The second approach only prints the booleans when the rule is violated because Sentinel short-circuits its evaluation and does not execute the print function when the value of `is_private and is_kms` is true. But if it is false, Sentinel has to evaluate the result of the print function in order to make its final determination for the rule. (While we know that the print function always returns true, Sentinel does not.)

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, your Sentinel policy will need to use the tfstate import or the state key of the tfplan import. This is an alias for the tfstate import but will retrieve the state from the plan.

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

We want to emphasize four points:

1. You need to refer to the "data" namespace under "tfplan.state" or use the tfstate import directly. If you do the latter, don't forget to import it at the top of your policy.
2. Instead of referring to the "applied" map of a data source, you will refer to its "attr" (attribute) map. So, a policy testing a data source with attribute "balance" would refer to "d.attr.balance" instead of "d.applied.balance".
3. Finally, while Sentinel allows a policy to directly refer to "tfplan.data" and then use something like "d.applied.balance", this generally won't work as desired because the applied object will be empty for many data sources, especially when you do the first run against a workspace. This causes the policy to pass when it should not. This behavior of Sentinel is counter-intuitive because the state is actually empty when executing the first run against a workspace. However, it is important to understand that "tfstate.data.*.attr" refers to the provisional state **after** the plan in the same way that "tfplan.resources.*.applied" refers to the expected state **after** the apply.
4. Terraform evaluates data sources during the refresh operation called by Terraform plan if all the arguments of the data source can be determined at that point in time. Otherwise, the

data source is not evaluated until the apply is done. Sentinel's tfstate import will therefore only have access to data from a data source if the data source was evaluated during the plan that preceded the policy check or during a previous run.

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 create 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: Create and test 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 you selected but fails if any certificate has a domain that is not a subdomain of it. For instance, here at HashiCorp, our policy should pass for domains such as "docs.hashicorp.com" but fail for domains such as "docs.hashycorp.com".

For this exercise, your Terraform code will have to use a data source to retrieve one or more specific ACM certificates. **These have to already exist within your AWS account.** Since your code will only retrieve one or more data sources, there will be nothing to apply. You do not need to create a new ACM certificate with your Terraform code. If you do not have an AWS account or do not have ACM certs in it that you can use, you can use this mock with the Sentinel Simulator.

We recommend that you set the AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_DEFAULT_REGION environment variables so that you do not need to include an "aws" provider configuration block in your Terraform code. Using environment variables is also more secure than embedding them in your code or in tfvars files. You’ll need to set the environment variables on the Variables tab of your Terraform workspace even if using the Terraform CLI since environment variables set in your shell for the CLI are not passed over to Terraform where the run will be executed. Please be sure to mark your AWS keys as sensitive so that no one else can see them.

Your Sentinel policy will need to use the state key of the tfplan namespace as described in the previous section.

Hint: One way of writing the policy uses the Sentinel matches operator. A second way uses the Sentinel strings import. If you use the Sentinel matches operator, 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 acm-certs directory and the restrict-acm-certificate-domains.sentinel policy, and the require-acm-certificate-domains test cases directory in the solutions repository. We give two ways of formulating the required condition.

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. Specifically, you would print something like `r.applied` (or `d.attr` for a data source) inside your rule that is evaluating the 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 a "tags" map that has two atributes.

```
  + 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. 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 an "all" or "any" 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 `all` and `for` loops. Use `all` loops inside a rule to evaluate some condition against all members of a list. Use `for` loops inside a function or in the main body of your policy to examine or process all members of a list.

Here is an example of a rule which prints all members of the subject_alternative_names list for all instances of the aws_acm_certificate resource. Note that we only use one variable after as when iterating over lists. The other `all` loops in this example use two variables because they are iterating over maps which have keys and values.

```
print_acm_sans = rule {
  all acm_certs as _, certs {
    all certs as index, c {
      all c.applied.subject_alternative_names as san {
        print( "List member in Rule:", san)
      }
    }
  }
}
```

You could use for loops outside a rule that would print out the same list by simply changing `all` to `for`. Additionally, if you knew that you had exactly 3 SANs in your list, you could also write `for` loops like this:

```
for acm_certs as _, certs {
  for certs as _, c {
    for [0, 1, 2] as num {
      print("List member:", c.applied.subject_alternative_names[num])
    }
  }
}
```

This illustrates that you can refer to a specific member of a list using the `<list>[n]` format in which `<list>` is the name of the list and n is the index of the member. Don't forget that lists are indexed starting with 0. (This code is probably not very realistic, but we wanted to show the syntax for accessing a specific member of a list.)

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 = rule {
  all acm_certs as _, certs {
    all certs as index, c {
      all c.applied.tags as k, v {
        print( "Map in Rule: 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. You could use for loops outside a rule that would print out the same map by simply changing `all` to `for`.

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 rule (in which we have shown the second way of referencing the key in a comment):

```
restrict_acm_ttl = rule {
  all acm_certs as _, certs {
    all certs as index, c {
      c.applied.tags.ttl is "24"
      #c.applied.tags["ttl"] is "24"
    }
  }
}
```

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":

```
for vms as address, r {
  if r.applied.storage_image_reference[0].publisher is not "RedHat" {
    print("VM", address, "has publisher",
          r.applied.storage_image_reference[0].publisher,
          "that is not allowed")
    validated = false
  }
}
```

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 must be computed 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`, 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 rule that needs to check 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 assures 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 write 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: Write and test a Sentinel policy that requires all Google compute instances provisioned by Terraform Enterprise 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.

You will need a GCP account to actually run a plan against your Terraform code and generate mocks. If you do not have one, you can use this mock with the Sentinel Simulator.

You can use the gcp_compute_instance code from the "Step 1" section of this guide. You will need to set the gcp_project and image Terraform variables and the GOOGLE_CREDENTIALS environment variable for your workspace if you do use that code. Be sure to mark your GOOGLE_CREDENTIALS environment variable as sensitive so it will not be visible to anyone else.

Hints: If you're not sure what attributes of the google_compute_instance to focus on in your Sentinel policy, please look at the code that creates the resource or review the google_compute_instance resource page. You should also look at a plan log from running your code or at the output of printing a single instance of the resource in a first iteration of your policy.

Extra Credit: It is possible to set the image for a Google VM in the google_compute_disk resource. Add a second rule to your Sentinel policy to control the image in that resource too.

Solution: See the gcp-instance directory and the restrict-gcp-instance-image.sentinel policy, and the restrict-gcp-instance-image test cases directory 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 (or the state key of the tfplan import). This section gives an example of using the tfconfig import, which can inspect the Terraform configuration files that are run within a Terraform workspace. Keep in mind that the tfconfig 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 write 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:

```
for resources as address, r {
  for r.provisioners as p {
    if p.type is "remote-exec" {
      print(address, "has remote-exec provisioner")
      validated = false
    }
  }
}
```

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 again define one function that retrieves certain resources, define a second function that validates them, and evaluate the results of invoking the latter in our main rule.

But there are some differences. The first and most obvious is that we import "tfconfig" instead of "tfplan". A second difference is that the find_resources_from_config function only has two for loops instead of three. The reason for this is that we are **not** iterating over the instances of each named resource. Recall that the find_resources_from_plan function that we used earlier had a third loop because each named resource could have a count metadata attribute which might result in multiple instances of that resource being created in the plan. *But instances don't exist in Terraform code within Terraform configurations!* So, we only need to iterate over the modules and named resources within the tfconfig namespace. A third difference is that we loop over the provisioners inside the validate_provisioners function.

Note that if you want to refer to attributes of resources evaluated with the tfconfig import other than provisioners, you need to use the config or references keys. The first is used with Terraform 0.11 regardless of whether the attributes are hard-coded or interpolated, but it can only access hard-coded values in Terraform 0.12. To access interpolated values in Terraform 0.12, the references key must be used instead of the config key.
Here are examples showing how you would reference the instance_type of an EC2 instance using the tfconfig import:

```
# Use with Terraform 0.11 and with hard-coded values in Terraform 0.12
print( "instance type:", r.config.instance_type)
# Use with interpolated values in Terraform 0.12
print("instance type:", r.references.instance_type)
```

This code won't show the actual values of "instance_type" such as "t2.micro" or "m4.large" that the EC2 instances will ultimately have unless those specific strings were assigned to the instance_type attribute. Instead, they will show whatever was assigned in the Terraform code; so, if your code had instance_type = "${var.client_instance_type}", the policy will print out ${var.client_instance_type}.

Additional Example of Using the Sentinel Simulator

In the final section of this guide, we provide an additional example of using the Sentinel Simulator with a mock of the tfconfig import. We also show how to use the Simulator's apply command.

To make the following example and exercise easier for you, we have generated two mocks of the tfconfig import:

- mock-tfconfig-fake-modules.sentinel was generated from a plan for a Terraform configuration that creates two instances of a fake module that creates a file containing the name set in a variable passed to the module.
- mock-tfconfig-azure-modules-test.sentinel was generated from a plan for a Terraform configuration that uses two Azure modules to create an Azure network and a VM.

We will give an example of using the first and have you use the second in Exercise 5.

To use the first mock with the Sentinel Simulator, do the following:

1. Create an empty directory called fake-modules-sentinel and navigate to it in a shell.
2. Create a mocks directory underneath the fake-modules-sentinel directory.
3. Download the mock-tfconfig-fake-modules.sentinel file to the mocks directory.
4. Create a Sentinel Simulator configuration file called sentinel.json in the fake-modules-sentinel directory that tells the simulator to load the mock:

```
{
  "mock": {
    "tfconfig": "mocks/mock-tfconfig-fake-modules.sentinel"
  }
}
```

5. Create a Sentinel policy called check-module-versions.sentinel in the fake-modules-sentinel directory that prints out the source and version of each module loaded directly from the root module of any Terraform configuration and requires the version to be "1.0.1":

```
import "tfconfig"

# Require all modules directly under root module
# to have version 1.0.1
check_module_versions  = func() {
  validated = true
  for tfconfig.modules as _, m {
    if m.version is not "1.0.1" {
      print("Module with source", m.source, "has barred version",
m.version)
      validated = false
    } else {
      print("Module with source", m.source, "has allowed version",
m.version)
    }
  }
  return validated
}

# Main rule that checks results of validation function
versions_validated = check_module_versions()
main = rule {
  versions_validated
}
```

6. You can now use the simulator to test your policy against the mock by running the command "sentinel apply check-module-versions.sentinel". You should see one line of output with "Pass" written in green.
7. You can make the simulator print a full trace more like what you would see if using the Sentinel policy in TFE by adding the "-trace" option before the name of the policy. In this case, the command would be "sentinel apply -trace check-module-versions.sentinel". You should see output like this (with some extra text that we have trimmed):

```
Pass

Print messages:

Module with source github.com/rberlind/terraform-local-fake has allowed version 1.0.1
Module with source github.com/rberlind/terraform-local-fake has allowed version 1.0.1

TRUE - check-module-versions.sentinel:20:1 - Rule "main"
```

The Sentinel Simulator also lets you run multiple tests in an automated fashion. To do that with the above policy and mock, do the following:

1. Create a directory called test under your current directory.
2. Navigate to that directory and create one under it called "check-module-versions". You must use this exact name since Sentinel is opinionated about the naming of test directories.
3. Copy the mock-tfconfig-fake-modules.sentinel file from the mocks directory to the check-module-versions directory, but change the name to mock-tfconfig-pass.sentinel.
4. Make a copy of the new file in the check-module-versions directory and call it mock-tfconfig-fail.sentinel.
5. Edit mock-tfconfig-fail.sentinel and change the versions of the two modules to "1.0.0"
6. Create a new test file called pass.json in the same directory with the following text which

configures a test to use the mock that will pass and also tells the simulator that the main rule should pass for this test:

```
{
  "mock": {
    "tfconfig": "mock-tfconfig-pass.sentinel"
  },
  "test": {
    "main": true
  }
}
```

7. Create a second test file called fail.json in the same directory with the following text which configures a test to use the mock that will fail and also tells the simulator that the main rule should fail for this test:

```
{
  "mock": {
    "tfconfig": "mock-tfconfig-fail.sentinel"
  },
  "test": {
    "main": false
  }
}
```

8. Navigate up two levels to the directory containing your check-module-versions.sentinel policy.
9. Now run both tests with the command "sentinel test". You should see the following output:

```
PASS - check-module-versions.sentinel
  PASS - test/check-module-versions/fail.json
  PASS - test/check-module-versions/pass.json
```

10. Note that both tests passed even though the second mock has the wrong version of the module. This happens because our second test said that the main rule should fail and it did. So, everything worked as expected.

Exercise 5

In this exercise, you will write a Sentinel policy that requires all Azure modules to come from the Private Module Registry (PMR) of your organization on your Terraform server. You will use the mock-tfconfig-azure-modules-test.sentinel tfconfig mock that we listed in the previous section.

Challenge: Write 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 Simulator to test your policy using both the apply and test commands.

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 not write any Terraform code. Instead, you will only write a Sentinel policy and then use the Sentinel Simulator to test it against the mocked data we have provided. The mock file actually pulls both modules from the public Terraform Module Registry, so your test with Sentinel will fail until you edit the source of both modules in the mock to match what you use in your policy.

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 TFE server running at https://app.terraform.io would have source "app.terraform.io/Cloud-Ops/network/azurerm".

After writing your policy and a Sentinel configuration file, run the `sentinel apply` command against your policy and the mock file. Then set up two test cases like we did above, create passing and failing versions of the mock file, and run the `sentinel test` command. Make sure that your Sentinel policy passes when all modules come from your PMR but fails if any module comes from a different source.

Hint: You can use the strings.has_prefix function or the matches operator to validate that all modules are in your PMR.

Solution: See the require-modules-from-pmr directory and the require-modules-from-pmr.sentinel policy 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, and tfstate 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 a Terraform configurations and workspaces, running plans and generating mocks against them, writing Sentinel policies, and testing of these policies with the Sentinel Simulator 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 Simulator with a tfconfig mock so that you can automate your testing of your Sentinel policies.

HashiCorp