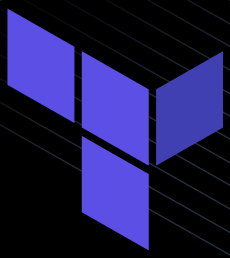

HashiCorp

Terraform

Writing and Testing Sentinel Policies for Terraform Enterprise

Contents

Introduction

This guide provides guidance for writing and testing Sentinel policies for Terraform Enterprise (TFE). 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 Enterprise 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 TFE, describes methods of testing them, and lays out a general methodology for creating Sentinel policies for Terraform. It also covers useful Sentinel operators, functions, and other concepts, and shows you how to use the Sentinel Simulator to test your policies. It includes some examples and five exercises that will give you experience writing Sentinel policies for TFE yourself.

This guide lays out a general methodology for creating Sentinel policies for Terraform and provides some examples and exercises. There are also many examples of Sentinel policies in the terraform-guides repository which are organized by cloud (AWS, Azure, GCP, and VMware).

Types of Sentinel Policies for Terraform Enterprise

There are essentially three types of Sentinel policies for Terraform Enterprise 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.

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 Sentinel Policies for Terraform Enterprise

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

1. You can manually test policies against actual Terraform code by using the Terraform Enterprise 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 TFE API.
3. You can use the Sentinel Simulator with the tfplan mock, the tfconfig mock, and the tfstate mock.

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 are not using Terraform at all and don't even need the Terraform binary to test your policies.

We will begin with the first method because many users creating Sentinel policies will already have Terraform code against which they want to test new policies. Additionally, it is the easiest method to learn and implement for those users even if they sometimes might need to create modified versions of their Terraform code to test both passing and failing of their policies. Finally, testing against actual Terraform code gives users more confidence that their Sentinel policies will behave as intended. We do include an example of the third method at the end of this guide.

The first method does have some negative aspects:

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 TFE 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 TFE 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 TFE 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.

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 Enterprise: 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 SaaS Terraform Enterprise server at https://app.terraform.io or access to a Private Terraform Enterprise (PTFE) server.
2. You will need to belong to the owners team within an organization on the Terraform Enterprise server you are using or have the Manage Policies organization permission for the organization.
3. If you want to use the TFE UI, you will also need to have an account in a VCS system such as GitHub in which you can create repositories to store your Terraform code. In this case, the organization will need a VCS Provider configured that is properly connected to your VCS account.
4. If you want to create Sentinel policies that create resources in a public cloud, you will need to have an account within that cloud. If you want to test the restrict-aws-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-google-compute-instance-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.

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. We will use the first method listed above for testing Sentinel policies; that means we will test them against Terraform configurations containing actual Terraform code by triggering runs against TFE workspaces using either the TFE UI or the Terraform CLI with the remote backend.

1. Create one or more Terraform configurations that create the resource your policy will restrict.
    a. Create a Terraform configuration that should pass the policy.
    b. If required, create one or more Terraform configurations that should fail the policy.
    c. In some cases, a single Terraform configuration can be used both to pass and fail a policy.
    d. If using the TFE UI, put your Terraform configurations in VCS repositories.
2. Create workspaces that use your Terraform configurations.
    a. Create one workspace for each configuration you created in step 1.
    b. If using a single configuration and workspace, add variables to it that can be given some values that will pass and some that will fail the policy.
3. Write a new Sentinel policy.
    a. Create a new policy set and add your workspaces to it.
    b. Create a new Sentinel policy and assign it to your policy set.
    c. Copy code from an existing policy.
    d. Retrieve all instances of some resource from all modules.
    e. Define one or more rules that restrict attributes of that resource.
    f. Combine the rules in your main rule.
    g. Save your policy.
4. Test your Sentinel policy against your workspaces.
    a. If using variables, set them so the policy should pass.
    b. Trigger a run against a workspace that should pass.
    c. If the policy failed or gave a hard error, revise the policy and repeat steps a and b until it does pass. (A hard error is one in which the Sentinel log complains about a syntax problem and refers to a specific line of the policy.)
    d. If using variables, set them so the policy should fail.
    e. Trigger a run against each workspace that should fail.
    f. If the policy passed or gave a hard error, revise until it fails without a hard error.

The order of the above steps is very natural to follow, especially for people who already have some Terraform code against which they want to test Sentinel policies.

While the above outline describes the creation of multiple Terraform configurations and workspaces that use them, in some cases a single workspace with a single configuration will suffice. Using Terraform variables which can be assigned different values often makes this possible. However, proper testing of some policies might require variations in your Terraform code itself; in these cases, more than one configuration will be needed.

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 TFE Sentinel policies because they help reduce costs by controlling the size of VMs in clouds.

Step 1: Create Terraform Configurations that Create the Resource

Let's create a Terraform configuration that will create a google_compute_instance resource so that we can test a Sentinel policy against it. In this case, we can actually create a single configuration and use a Terraform variable called machine_type to set the machine_type attribute of the google_compute_instance with different values for each run. When we set it to "n1-standard-1", "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 policy. 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 3:

```
variable "gcp_credentials" {
  description = "GCP credentials needed by google provider"
}

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" {
  credentials = "${var.gcp_credentials}"
  project     = "${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
    }
  }
}
```

You will want to put this code in a single main.tf file. If you will be using the TFE UI, then upload this code to a VCS repository. If you will be using the Terraform CLI with the remote backend configured, you will also want to include code like the following either in your main.tf file or in a separate file which could be named something like backend.tf:

```
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 TFE 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 TFE server, change the hostname field to its URL.

Step 2: Create Workspaces that Use Your Terraform Configurations

After creating one or more Terraform configurations in Step 1, you need to create a workspace for each of them. To do this, follow the instructions for creating workspaces. If you use the TFE UI, then point the workspaces against the VCS repositories containing your Terraform configurations. (Note that your TFE organization must already have a VCS connection set up that points at the VCS system managing these repositories.) If you use the Terraform CLI with remote backend, then select "None" as the source for your workspaces so that they are not linked to any VCS repositories.

For our example, you only need to create a single Terraform workspace since you only created one configuration. After creating your workspace, be sure to set the gcp_project, gcp_credentials, and machine_type Terraform variables on the Variables tab of your workspace. Start out by setting the machine_type variable to "n1-standard-1". You can also set a value for the instance_ name variable if you don't want to use the default value of "demo". 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. The gcp_credentials variable should be set to the contents of your local copy of the Google Cloud service account file containing your GCP credentials. 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 gcp_credentials variable as sensitive so that other people cannot see or copy what you paste into it.

Step 3: Write a New Sentinel Policy

Here are the sub-steps of Step 3:

a) Create a new policy set:

TFE policy sets control which workspaces Sentinel policies are applied against. While you could assign your new policy to a global policy set that would be applied to all workspaces, it is a good idea to create a policy set just for testing your new policy and to use an organization that does not have any global policy sets. That way, when you test your policy against your workspaces, no other policies will be tested against your Terraform code. This makes your life easier since you won't have to read through the results of any other policies to find the results of your new policy.

When creating your new policy set, set the scope to "Policies enforced on selected workspaces". Add the workspace you created in Step 2 to your new policy set. Do not add any policies to it at this time. We suggest giving the policy set the same name as the policy you will be adding to it.

b) Create a new Sentinel Policy and assign it to your policy set:

To create a new policy that restricts the size of Google compute instances, simply follow the directions in the Managing Policies documentation. You will create the policy, give it a name and a description, and set the enforcement mode: advisory, soft mandatory, or hard mandatory. Violations of *advisory* policies log violations without preventing a subsequent apply from being done. Violations of *soft mandatory* policies prevent an apply from being done unless an authorized user overrides the violation. Violations of hard mandatory policies prevent an apply from being done and cannot be overridden. We recommend using the *hard mandatory* enforcement mode when testing new Sentinel policies to avoid having to decline overriding of failed tests.

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-google-compute-instance-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.

After naming the policy, be sure to assign the new policy to the policy set you created above.

Note that it is also possible to upload a policy to your organization with TFE's Policy Sets and

Policies APIs or using the Terraform Enterprise Provider, which is a Terraform provider that can manage TFE resources. An example of using the TFE provider to manage policy sets and policies is in the hashicorp/tfe-policies-example repository.

c) Copy code from an existing policy:

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-aws-instance-type policy.

Each Sentinel policy intended to restrict attributes of resources or data sources will use the Sentinel tfplan import by including this statement at the top:

```
import "tfplan"
```

So, be sure to include that when you copy code from an existing Sentinel policy.

d) Retrieve all instances of some resource from all modules:

When picking a policy, be sure to pick one that starts with a function that retrieves all instances of a resource from all modules within the Terraform configuration. 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. For instance, if you create a policy to require all S3 buckets to use KMS encryption, but only find S3 buckets in the root module, S3 buckets could be created in nested modules without KMS encryption.

Here is an example function that retrieves all instances of the aws_instance resource from all modules:

```
get_aws_instances = func() {
  instances = []
  for tfplan.module_paths as path {
    # Next two lines split here for readability in this guide
    instances += values(
    tfplan.module(path).resources.aws_instance) else []
  }
  return instances
}
aws_instances = get_aws_instances()
```

We start by creating an empty list called `instances`. We then loop through all modules in the current plan with a for loop, referring to each module found as `path`. We then add all instances of the aws_instance resource from the current module to the instances list. Note the important inclusion of the else operator in "`else []`"; this makes sure that we do not add the Sentinel value `undefined` to the `instances` list if the current module does not have any aws_instance resources. Doing that would make the entire list `undefined` and break the policy. Adding "`else []`" adds an empty list to the current instances list which leaves it unchanged. At the end of the function we return the list of all aws_instance resources. Finally, we call the function and store the results in the `aws_instances` list outside the function.

When copying a function like this from an existing policy, you will generally want to change the name of the function ("get_aws_instances" in this case), the name of the lists ("instances"), and the resource type being targeted ("aws_instance"). So, if you wanted to create a Sentinel policy that restricted the google_compute_instance resource, you would end up creating a function like this:

```
get_google_instances = func() {
  instances = []
  for tfplan.module_paths as path {
    # Next two lines split here for readability in this guide
    instances += values(
    tfplan.module(path).resources.google_compute_instance) else []
  }
  return instances
}
compute_instances = get_google_instances()
```

In this case, we kept instances as the name of the list because that works for both aws_instance and google_compute_instance.

e) Define one or more rules that restrict attributes of that resource:

If we look more closely at the restrict-aws-instance-type policy, we see that it defines an `allowed_types` list and an `instance_type_allowed` rule. While all Sentinel policies will have at least one rule, only some policies will have lists like the `allowed_types` list. Other policies will simply include a rule which requires that some attribute exists, equals a single value given in the rule, or is contained within a list of values embedded in the rule itself.

```
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 rule that actually restricts the size of the aws_instance resource:

```
instance_type_allowed = rule {
    all aws_instances as _, instances {
      all instances as index, r {
         r.applied.instance_type in allowed_types
      }
    }
}
```

Sentinel rules must return a boolean value, `true` or `false`. We actually loop through the items of the `aws_instances` list (returned by calling the `get_aws_instances`() function) in two `all` loops. The first loop gives us each named occurrence of the aws_instance resource within our Terraform code. The second loop iterates through all instances of each named occurrence. This is required because any named occurrence of a Terraform resource can create multiple instances of the resource based on the count metadata attribute. Using two loops ensures that Sentinel evaluates the policy against all instances of all named resources of the specified type, at least until it finds one that breaks the rule, at which point it returns `false`.

Using the all expression is what ensures that the rule will only return true if all instances of aws_instance have allowed sizes. If we wanted to make sure that at least one instance satisfied some property, we would use the "any" expression instead of the "all" expression in both loops. This is described on the same Sentinel documentation page that describes the "all" expression.

Additionally, since the aws_instances list is actually a list of key/value maps, each loop actually iterates through two lists: the keys and the values. After each occurrence of "as" in a loop statement (which could use `all` or `any`), we see two items; the first will contain the keys while the second will contain the values. However, we often only care about the values. Sentinel emulates the Go programming language in allowing us to use "_" to indicate that we won't need to refer to one of the two lists and therefore don't want to name it. In our example, the use of "_" indicates that we are not naming the list of keys returned by the first loop. We also could have used "_" instead of "index" in the inner loop since we never referred to the index list.

In this policy, we use one of Sentinel's set operators, "`in`", to check if the instance_type attribute of each aws_instance is in the `allowed_types` list. So, we have `r.applied.instance_type` in `allowed_types` inside the inner loop. The other set operator is "`contains`" which checks if a list contains a specific value.

We mentioned in the introduction that TFE 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.instance_type`" gives us the instance_type of each instance of the aws_ instance resource that would exist if the apply were allowed to run.

Now, let's think about how we can modify that code 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, the specific resource mentioned in the rule, and the attribute (`instance_ type`) mentioned in the 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 rule to look like this:

```
machine_type_allowed = rule {
    all compute_instances as _, instances {
      all instances as index, r {
         r.applied.machine_type in allowed_types
      }
    }
}
```

Note that we have changed the rule name from `instance_type_allowed` to `machine_ type_allowed`, changed `aws_instances` to `compute_instances` to match the name of the list returned by the `get_google_instances` function, and changed `instance_type` to `machine_type`.

f) Combine the rules in your main rule:

We are almost done modifying our policy, but we need to make one more change. The restrict-aws-instance-type policy that we started with had a `main` rule that looked like this:

```
main = rule {
  (instance_type_allowed) else true
}
```

This main rule requires the instance_type_allowed rule that was also in that policy to be true. Since we changed the name of that rule to `machine_type_allowed`, we need to change our main rule to use that. Making this change gives us:

```
main = rule {
  (machine_type_allowed) else true
}
```

That completes the modifications to our policy. However, we want to talk about the `else true` in the main rule. This is similar to what we had in our functions that returned all instances of some resource; it protects against the possibility that the rules the main rule refers to might give the `undefined` value. Including `else true` converts undefined into `true` so that the main rule returns `true`. This is almost always done in the main rule of Sentinel policies. Otherwise, policies meant to restrict one resource type might fail when run against Terraform configurations that do not even create that resource.

We should also note that it is common to combine multiple rules in a policy's main rule by using Sentinel's logical operators such as "and", "or", "xor", and "not". You can see this in some of the Sentinel examples in the terraform-guides repository including this one. The most common choice is to use "and" to require that all the other rules in the policy are true.

g) Save your policy:

Now that you have finished creating your new Sentinel policy to restrict the size of Google VMs, you can save it by clicking the purple "Create Policy" button at the bottom of the Terraform Enterprise Sentinel policy editor. Don't forget to assign your new policy to the policy set you created earlier.

Your final policy should be called something like "restrict-google-compute-instance-machine-type" and should look like this:

```
import "tfplan"

get_google_instances = func() {
```

```
  instances = []
  for tfplan.module_paths as path {
    # Next two lines split here for readability in this guide
    instances += values(
    tfplan.module(path).resources.google_compute_instance) else []
  }
  return instances
}
compute_instances = get_google_instances()

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

machine_type_allowed = rule {
    all compute_instances as _, instances {
      all instances as index, r {
         r.applied.machine_type in allowed_types
      }
    }
}

main = rule {
  (machine_type_allowed) else true
}
```

Step 4: Test Your Sentinel Policy Against Your Workspaces

After creating your Terraform configurations in Step 1, your workspaces in Step 2, and your Sentinel policy in Step 3, you can begin to test your policy against your Terraform code. You will do this by starting runs against each workspace and examining the results of the Sentinel policy check. If using the TFE UI, click the "Queue Plan" button in your workspace, enter a comment, and click the second "Queue Plan" button. If using the Terraform CLI with the remote backend, run the `terraform plan` command from the directory containing your Terraform configuration. (Note that you will have to run `terraform init` before running your first test against any configuration.)

Remember that the Sentinel policy will be checked after a plan is run against your workspace. If you set up your policy set as recommended above, only your new policy should be checked against your workspace. If not, other policies might also be checked and you will need to read through the results of those policies until you find the results for your policy.

The entire Terraform plan will be run before any Sentinel policies are checked. Note that the results of Sentinel policies are completely independent of each other.

After all Sentinel policy checks have been run against your workspace, it is possible that the policy check section of the run page will automatically collapse. To view the Sentinel policy checks, just click on the policy check section. You will then see a list of all Sentinel policies that were checked along with "passed" or "failed" indications. Further down in the expanded section, you will see the log with details of all Sentinel policy checks executed during the run. You can expand that log and even make it occupy the full screen within your browser tab. Of course, you can also then minimize and collapse the Sentinel policy log.

The result for our policy will look like this if it failed:

```
## Policy 1: restrict-google-compute-instance-machine-type.sentinel (soft-mandatory)

Result: false

FALSE - restrict-google-compute-instance-machine-type:28:1 - Rule "main"
  FALSE - restrict-google-compute-instance-machine-type.sentinel:21:5 - all compute_
instances as _, instances {
    all instances as index, r {
        r.applied.machine_type in allowed_types
    }
}

FALSE - restrict-gcp-machine-type.sentinel:20:1 - Rule "machine_type_allowed"
```

The result for our policy will look like this if it passed:

```
## Policy 1: restrict-google-compute-instance-machine-type.sentinel (soft-mandatory)

Result: true

TRUE - restrict-google-compute-instance-machine-type:28:1 - Rule "main"
  TRUE - restrict-google-compute-instance-machine-type.sentinel:21:5 - all compute_
instances as _, instances {
    all instances as index, r {
        r.applied.machine_type in allowed_types
    }
}

TRUE - restrict-gcp-machine-type.sentinel:20:1 - Rule "machine_type_allowed"
```

It is also possible that you will see a hard failure with a message indicating that an error occurred. For instance, you might see something like this:

```
An error occurred: restrict-s3-buckets.sentinel:34:10: only a list or map can be
indexed, got undefined
```

In this case, the policy could not be run because of a syntax error. You will need to modify your policy to fix the problem.

A complete, successful test process for any Sentinel policy requires that it pass when run against Terraform configurations which satisfy its main rule and that it fail when run against Terraform configurations that violate its main rule. In most cases, the main rule will combine all other rules with the logical operator "and" between them; this means that the policy should pass when run against Terraform configurations which satisfy all of its rules and that it should fail when run against Terraform configurations that violate one or more of its rules. However, as noted in Step 3(f) above, a main rule could use other logical operators to combine rules.

You need to make sure that one of your Terraform configurations tests the passing of all rules and that other Terraform configurations test the failure of each rule. So, if your policy had four rules, all of which are supposed to be true, you should test a configuration that would satisfy all four rules, one that fails the first rule, one that fails the second rule, one that fails the third rule, and one that fails the fourth rule. While that implies that you would need five configurations in this case, you can often use Terraform variables to reduce the number of configurations, maybe even all the way down to just one.

Additionally, it is a good idea to write Terraform configurations that create more than one resource of the type you are testing. This allows you to test that your Sentinel policy is correctly looping through all instances of the resource and correctly failing if even one of them violates a condition.

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.

Before you begin the exercises, it's important to properly configure your environment:

If you connect your TFE workspaces to VCS repositories, we recommend that you put each one in its own repository to avoid triggering runs against multiple Terraform workspaces when you make changes to your code.

If you use the Terraform CLI with the remote backend, then you do not need to put your Terraform configurations in repositories. However, you will still want to put your code for each exercise in its own directory so that you can run the code for the different exercises separately.

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.

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 created by Terraform Enterprise to the following choices: Azure, Kubernetes, GitHub, and AppRole.

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.

A single Terraform configuration and associated workspace will suffice to test your policy if you use Terraform variables to determine the type and path of the Vault auth method and change their values between runs. We do recommend you set the path as well as the type variables so that you do not conflict with others using the same Vault server. If you use the TFE UI, set your variables on the Variables tab of your workspace. If you use the Terraform CLI, add the values of your variables to a file with a name like "vault.auto.tfvars" that is in the same directory as your Terraform configuration.

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. You can combine some that pass, some that fail, and a combination of some that pass and some that fail.

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 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 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 and the restrict-vault-auth-methods policy 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 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 a double "all" loop such as the one in our example policy, Sentinel stops evaluating instances as soon as one violates the condition limiting the VM type/size.

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.

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. For each test, you will only need to run a plan and then let TFE check your Sentinel policy against it. After that, you can discard the run without doing an apply.

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.

It will probably be easier in this case to use two repositories and two workspaces. The Terraform code in the first should include the PGP key while the code in the second should not.

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.

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

Using Print in Sentinel Policies

A very useful way to debug problems with your Sentinel policies is to add calls to the built-in print function. You can add these in your own functions and in your rules. 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.)

Including the print function in all of your 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!*

As an example, if you wanted to print the kms_key_arn attribute of each instance of the aws_lambda_function resource being provisioned in a rule which requires that the KMS key of the lambda functions not be blank, you could use the following rule:

```
require_kms_key = rule {
  all lambda_functions as _, instances {
    all instances as index, r {
      print("kms_key_arn: ", r.applied.kms_key_arn) and
      (r.applied.kms_key_arn is not "")
    }
  }
}
```

When you include the print function in your 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. In the example we just gave, you could try printing `r.applied` instead of `r.applied.kms_key_arn`.
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. (Sentinel stops processing a rule

as soon as it determines that it is `true`, `false`, or `undefined`.)

In policies written with "all" loops, you can place the print function before or after the condition in your rule, with the choice depending on which instances you want to print. If you want to include all instances up to the first violator of the rule, place the print function before the other conditions and add the "and" operator after it as was shown above. The last one printed is then the culprit. If you only want to print out the first violator of the rule, place the print function after the other conditions and add "or not" before it. We include "not" so that the final boolean expression is still false.

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

```
require_kms_key = rule {
  all lambda_functions as _, instances {
    all instances as index, r {
      (r.applied.kms_key_arn is not "") or not
      print("kms_key_arn: ", r.applied.kms_key_arn)
    }
  }
}
```

The second approach only prints out the first instance that violates the rule because Sentinel short-circuits its evaluation and does not execute the print function when the rest of the rule is true. But if the rest of the rule 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, there are some significant differences. The first is that your Sentinel policy will need to use the state key of the tfplan namespace. This is an alias for the tfstate import but will retrieve the state from the plan. Alternatively, you could use the tfstate import directly.

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 three 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 "tfplan.state.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.

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 Enterprise 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.
You can use a single workspace by setting suitable variables in your code and the workspace. 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.

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.)

Extra Credit: Write your Sentinel policy so that it only prints out the domain for the first invalid certificate.

Solution: See the acm-certs directory and the restrict-acm-certificate-domains.sentinel policy in the solutions repository. We give two ways of formulating the required condition and illustrate printing before and after the 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:

```
+ 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. 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 a Sentinel rule which requires the publisher to be "RedHat":

```
vm_publisher_allowed = rule {
    all vms as _, instances {
      all instances as index, r {
         r.applied.storage_image_reference[0].publisher is "RedHat"
      }
    }
}
```

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:

```
require_versioning_ = rule {
    all s3_buckets as _, instances {
      all instances as index, r {
        types.type_of(r.applied.versioning) is "list" and
        r.applied.versioning[0].enabled
      }
   }
}
```

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.

A single Terraform configuration and associated workspace will suffice to test your policy if you use a Terraform variable to determine the image and change its values between runs. You could even use the gcp_compute_instance code from the "Step 1" section of this guide. You will also need to set gcp_project and gcp_credentials Terraform variables for your workspace if you use that code.

If you use the TFE UI, set your variables on the Variables tab of your workspace. Be sure to mark your gcp_credentials variable as sensitive so it will not be visible to anyone else. If you use the Terraform CLI, add the values of your variables to a file with a name like "gcp.auto.tfvars" that is in the same directory as your Terraform configuration.
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 in the solutions repository.

Using the TFE Sentinel tfconfig Import

So far, we've discussed several examples of using the tfplan import with TFE and one example of using the tfstate import, which we used indirectly via 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 TFE workspace.

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 TFE 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}"
    }
  }
}
```

We are showing the Terraform code in order to make the following Sentinel policy that blocks remote-exec provisioners from being run easier to understand. Note that the provisioner is a block under the null resource. Any resource can have zero, one, or multiple provisioner blocks.

Here is our Sentinel policy:

```
import "tfconfig"

get_null_resources = func() {
  nrs = []
  for tfconfig.module_paths as path {
    nrs += values(
    tfconfig.module(path).resources.null_resource) else []
  }
  return nrs
}

null_resources = get_null_resources()

only_local_exec_provisioners = rule {
  all null_resources as _, nr {
    all nr.provisioners as p {
      print( "provisioner type: ",p.type) and
```

```
      p.type != "remote-exec"
    }
  }
}

# Main rule that requires other rules to be true
main = rule {
  (only_local_exec_provisioners) else true
}
```

You'll notice that the overall pattern of this policy is very similar to the ones we wrote to restrict resources. We again declare a function that retrieves certain resources, invoke that function, define a rule that uses `all` loops to evaluate the matching resources, and define a main rule that requires the other rule to be true.

But there are some differences. The first and most obvious is that we import "tfconfig" instead of "tfplan". The second and more important difference is in the use of the `all` loops: There are still two of them just like in all our other examples, but you might have expected three of them to account for diving into the resource and iterating over all of its provisioners.

However, we are not iterating over the instances of each named null resource. Recall that we did the second loop in our other policies because each named resource and data source could have a count metadata attribute which might result in multiple instances of that resource or data source being created in the plan, apply, and ultimately in the state. *But instances don't exist in Terraform code within Terraform configurations!* So, we only need to iterate over the named resources within the tfconfig namespace to reach the point at which we can iterate over provisioners associated with the resources.
For reference, we also share a snippet of Sentinel policy code that would show what was passed to the `instance_type` attribute of each `aws_instance` resource:

```
print_ec2_instance_type = rule {
  all aws_instances as _, r {
    print( "instance type: ", r.config.instance_type)
  }
}
```

In this case, we only have a single `all` loop. Note that you have to use `r.config` in this case to access the attributes of the resource. That is because the actual resource map visible to Sentinel in the tfconfig import has all of the resource attributes under a nested map called "config".
We did not need "config" when iterating through the provisioners above because that policy examined the type attribute of each provisioner which is outside the config map..

This policy 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, it 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}`.

We close this section by emphasizing again that you should not include an all loop for instances when using the tfconfig import.

Using the Sentinel Simulator with TFE Sentinel Mocks

In the final section of this guide, we're going to illustrate how you can use the Sentinel Simulator with mocks of the TFE Sentinel imports. A mock simulates what the output of a Terraform plan would produce. Since there are three TFE Sentinel imports (tfplan, tfstate, and tfconfig), there are three corresponding mocks. We'll use the tfconfig mock.

The method of using the Sentinel Simulator with the tfplan, tfconfig, and tfstate mocks has a steeper learning curve than testing policies directly against Terraform configurations in TFE since creators of policies need to learn how to use the tool; but this method does have some advantages over testing Sentinel policies against actual Terraform code in TFE:

1. Each test of a policy with the Sentinel Simulator with mocks is faster since the test will not run a Terraform plan.
2. Using the simulator avoids having to discard runs against workspaces and avoids having those runs in your workspace history.

If you want to automate testing of policies that need frequent changes, then using the simulator is the best choice since each test will be faster and will not place extra load on your TFE server.

Using the simulator with mocks does has the potential risk that you might write your test or the mock incorrectly and end up with an invalid test of your Sentinel policy. In contrast, testing policies against actual Terraform code gives users more certainty that their Sentinel policies are really behaving as intended. In the near future, HashiCorp expects to release a mock generator that will make the creation of mocks for the three Terraform Sentinel imports less error-prone. At that point, using the Sentinel Simulator with mocks will become much easier.

The first thing you'll want to do is download the Sentinel Simulator for your OS from the Sentinel Downloads page. We used version 0.9.0 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>".

Terraform Enterprise makes it easy for users to generate the tfplan, tfconfig, and tfstate mocks against any plan they have run in the past seven days. In the TFE UI, you can select a run from a workspace, expand the plan, and click the "Download Sentinel mocks" button to download a tar file with the mocks. You can also use the TFE Plans API to download the mocks. If desired, you can then edit the mocks to change various values in them.

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 4.

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"
import "strings"

# Require all modules directly under root module
# to have version 1.0.1
check_module_version  = rule {
  all tfconfig.modules as _, m {
    print("source: ", m.source, ", version: ", m.version) and
    m.version == "1.0.1"
  }
}

# Main rule that requires other rules to be true
main = rule {
  (check_module_version) else true
}
```

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:

source:  github.com/rberlind/terraform-local-fake , version:  1.0.1
source:  github.com/rberlind/terraform-local-fake , version:  1.0.1

TRUE - check-module-versions.sentinel:14:1 - Rule "main"
  TRUE - check-module-versions.sentinel:7:3 - all tfconfig.modules as _, m {
    print("source: ", m.source, ", version: ", m.version) and
        m.version == "1.0.1"
    }
    TRUE - check-module-versions.sentinel:8:5 - print("source: ", m.source, ", version: ",
m.version)
    TRUE - check-module-versions.sentinel:9:5 - m.version == "1.0.1"
    TRUE - check-module-versions.sentinel:8:5 - print("source: ", m.source, ", version: ",
m.version)
    TRUE - check-module-versions.sentinel:9:5 - m.version == "1.0.1"

TRUE - check-module-versions.sentinel:6:1 - Rule "check_module_version"
  TRUE - check-module-versions.sentinel:8:5 - print("source: ", m.source, ", version: ",
m.version)
  TRUE - check-module-versions.sentinel:9:5 - m.version == "1.0.1"
  TRUE - check-module-versions.sentinel:8:5 - print("source: ", m.source, ", version: ",
m.version)
  TRUE - check-module-versions.sentinel:9:5 - m.version == "1.0.1"
```

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 TFE 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 Enterprise server. Then use the Sentinel Simulator to test your policy.

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.

For this exercise, you will not write any Terraform code. In fact, you won't even be using Terraform or TFE. 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 <TFE HOSTNAME>/<TFE 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".

Initially, you can run the "sentinel apply" command against your policy and the mock file. However, we encourage you to set up two test cases like we did above, create passing and failing versions of the mock file and configuring the "pass" test to satisfy the main rule and the "fail" test to violate it. 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 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 Enterprise. We discussed different types of Sentinel policies that use TFE's tfplan, tfconfig, and tfstate imports. We described three methods of testing TFE Sentinel policies and discussed their pros and cons. We laid out a basic four-step methodology for writing and testing new Sentinel policies that restrict attributes of resources. This methodology includes creating Terraform configurations and workspaces, writing Sentinel policies to test against them, and the actual testing of those policies.

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 results 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 you how to use the Sentinel Simulator with TFE mocks so that you can automate your testing of your TFE Sentinel policies.


HashiCorp