Terraform Security 101: Best Practices for Secure Infrastructure as Code

@bridgecrewio · Aug 25,2021 · 8 min read · 266 views · Originally posted on bridgecrew.io

#Terraform #iac #Kubernetes

Deploying and managing cloud resources is faster and easier than ever, and we have infrastructure as code (IaC) to thank for it. With IaC, tedious manual configurations and one-off scripts are things of the past. Instead, you manage infrastructure with code in much the same way you would applications and services. This infrastructure can be anything from servers and databases to networks, Kubernetes clusters, and entire application stacks.

Terraform by HashiCorp is a popular, multi-cloud IaC framework. It uses a declarative approach, meaning you define how you want infrastructure to look rather than the steps to reach that outcome. To make changes to infrastructure, such as adding more instances with the same configurations, you simply define the changes in the template and Terraform does the rest.

One of the other great things about Terraform is that it is modular. This makes it easier for teams to deploy and scale infrastructure with just a few lines of code. By modifying just a few variables, you have ready-made networking, storage, or compute workloads ready for deployment.

Getting started with Terraform

Before we dive into the security aspect of Terraform, let's start with some basics. Terraform lets us configure systems using a human-readable, declarative syntax. A basic configuration looks like this:

```
provider "aws" {
profile = "default"
region = "us-west-2"
}

resource "aws_instance" "myserver" {
ami = "ami-830c94e3"
instance_type = "t2.micro"
}
```

The first block determines what cloud provider we plan to use. In this case, it's AWS, but we can pick from any of the hundreds of providers supported.

The second block defines a resource, a group of systems treated as a logical unit. Resources in Terraform take two arguments—a resource type and a local name. In the example, the type aws_instance corresponds to one or more EC2 virtual machines.

To initialize a working directory containing Terraform configuration files, we'll run:

```
terraform init
```

This will parse the Terraform config and download the necessary plugins:

```
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v3.15.0...
- Installed hashicorp/aws v3.15.0 (signed by HashiCorp)

Terraform has been successfully initialized!
```

The second step (optional) is to examine the current state—the configuration settings stored locally or remotely of the infrastructure—and create an execution plan by running:

```
terraform plan
```

This command performs a series of tasks, ensuring that the Terraform state is up-to-date, spotting differences between the current and prior configuration states, and proposing what should happen next. For our example, it would print:

Kubernetes vs OpenShift: This Is What You Need To Know

3 months, 1 week ago by @qudems

Three Kubernetes Success Stories to Inspire Your Next Application

3 months, 2 weeks ago by @annie

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create


Terraform will perform the following actions:

# aws_instance.myserver will be created
+ resource "aws_instance" "myserver" {
+ ami = "ami-830c94e3"
+ instance_type = "t2.micro"
+ arn = (known after apply)

+ ebs_block_device {
...
}

+ ephemeral_block_device {
...
}

+ network_interface {
...
}

+ root_block_device {
...
}
}

Plan: 1 to add, 0 to change, 0 to destroy.
```

To create the resources, all we have to do is issue the following command:

```
terraform apply
```

Terraform will output the plan again and ask for confirmation:

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

At this point, you can see how Terraform improves upon the scalability and flexibility benefits of cloud-native environments by making them modular, machine-readable, and version-controllable. When done correctly, these benefits can also be leveraged to embed security guardrails earlier in your cloud-native application lifecycle. There are, however, security implications to using Terraform that we will learn about in the next section.

Securing Terraform

By design, IaC doesn’t present itself as an immediate risk or attack surface. But because IaC is governed by engineering and DevOps, security teams may often overlook it, instead focusing on monitoring cloud resources already in production (where risk is more immediately addressable).

For example, a publicly exposed S3 bucket might seem like more of an immediate risk than code that could result in exposed buckets if the right protective layers aren't in place. While there is some truth to that, you may be overlooking some benefits and risks of ignoring IaC's security implications.

Developing and managing infrastructure at scale is complex, and security and DevOps teams on their own often don't have the know-how, access, or tools they need to take on security.

As a result, it's easy for security measures to be missed and cloud resources to be incorrectly configured. For example, it is common for engineers and developers to:

- Use default configurations that haven't been optimized for security.
- Not enable logging, which makes it difficult to troubleshoot or assemble an audit trail.
- Use unencrypted databases that leave data vulnerable to corruption and exfiltration.
- Deploy insecure protocols (e.g., not using HTTPS).
- Leverage vulnerable microservices.

Simply put, IaC is cloud configuration defined and managed using code. This also means all security configuration must be defined in code. For example, to add versioning to an S3 bucket, you would need to define that in code.

This is what a private S3 bucket with versioning enabled looks like:

```
module "s3_bucket" {
source = "terraform-aws-modules/s3-bucket/aws"

bucket = "my-s3-bucket"
acl = "private"

versioning = {
enabled = true
}

}
```

If variables are left undefined or are misconfigured, resources may be publicly exposed in your production cloud environment, potentially introducing risk. It can be helpful to get a real-life example of what these misconfigurations might look like. TerraGoat is a learning and training project that demonstrates how common configuration errors can find their way into production cloud environments.

Here are a few best practices that we have to keep in mind when using Terraform.

Securely leverage Terraform modules

Developers are great at finding ways to simplify and automate some of their responsibilities. Leveraging open-source is one way to do so. In fact, one of the benefits of IaC is the creation and sharing of pre-built templates or modules, which makes it

easier and faster to get cloud services up and running.

There are pre-built sets of Terraform modules that provide a specific service, such as deploying a Virtual Private Cloud (VPC) and the associated security groups. Modules provide standard functionalities to help get you started quickly. Simply define some input variables, call the module, and leave the rest to Terraform.

There are two types of modules:

- Private modules: These are files that you write as part of your configuration. A module encapsulates reusable components that other people in your team can take advantage of.
- Public modules: These are third-party modules publicly available for reuse in places like GitHub or published on the Terraform Registry, which contains more than 4,000 modules with new modules added daily.

Although modules are key to running quickly with IaC, they aren't typically built and shared with a security-first mindset. Securing the use of modules lies on the user. Just like integrating any other open-source code into your codebase, you are responsible for adding all necessary security configurations yourself.

To illustrate just how important this is, we analyzed thousands of open-source Terraform modules in our State of Open Source Terraform Security Report. Our research found that around half of all open-source Terraform modules within the Terraform Registry contained misconfigurations. This research highlights the gap in how and where cloud security is being addressed. It also shows how much security has lagged and how big of an impact DevSecOps can have for securing cloud infrastructure.

Declare variables

You can declare variables in your modules to make them reusable. This is one of the most impactful benefits of IaC. Variables are also convenient to keep secrets, such as passwords and API keys, outside the code.

To use a variable, we must declare it first:

```
variable "password" {
description = "Password for the connection"
type = string
}
```

Variables declared with a default value are optional:

```
variable "aws_region" {
description = "AWS region to launch servers."
type = string
default     = "us-west-2"
}
```

Terraform comes with three base types: string, number, and bool. But we can also build complex structures by combining these data types. Terraform supports `list`, `map`, `set`, `tuple`, and `object`.

```
variable "availability_zones" {
type    = list(string)
description = "List of availability zones"
default = ["us-west-1a"]
}
```

After declaring a variable, you can supply its value in four different ways:

- Interactively while running `terraform apply`
- As a command line argument: `terraform apply -var="password=Sekret1"`
- In environment variables: `TF_VAR_password="Sekret1" terraform apply`
- In a separate file: `terraform apply -var-file="secrets.tfvars"`. These files should be kept secured and should never be checked into source control.

While declaring variables can be impactful, it also introduces added complexity. If a variable contains a misconfiguration, all modules that reference that variable will also be misconfigured. So, it's important that you have the proper tooling to scan variables. We'll get into that more in the next section.

Infuse automated scanning into development processes

Another benefit of using Terraform to define infrastructure is the ability to audit code for misconfigurations before any infrastructure is created. In this way, you can incorporate security into development processes earlier and prevent infrastructure issues (like opening an S3 bucket to the world) from being deployed to your running cloud environment.

Using open-source, static code analysis tools like Checkov, you can scan your Terraform templates and directories for misconfigurations without the added friction of integrating your own code. A tool like Checkov has hundreds of built-in policies that cover encryption, network, backup, and identity and access management (IAM) compliance and security configurations for AWS, GCP, and Azure cloud providers.

For consistency, automate IaC scanning in your continuous integration/continuous delivery (CI/CD) pipeline. This allows you to provide automated feedback as a part of a CI run and potentially block misconfigured code.

You can also be notified about infrastructure as code (IaC) misconfigurations and policy violations at the earliest possible moment within the DevOps lifecycle: when you are coding on your local workstation. You can get feedback directly in your integrated development environment (IDE) using Bridgecrew's Visual Studio Code (VS Code) extension.

Review the plan for security issues, too

With local or CI/CD scanning of your Terraform code, you get instant feedback on your modules and templates. But because of Terraform's dependency-driven nature, you might not be getting the full picture. In order to get a holistic view into what is actually being provisioned or changed, including variables being called, you may need to scan the Terraform plan output. But you can also analyze a misconfiguration even if it was sourced in a parameter that was defined on a different code block. This ensures that no risk goes unidentified.

SaaS platforms such as Bridgecrew, make this easy with native integrations. One such integration is with Terraform Cloud. With Bridgecrew's continuous policy enforcement and security feedback for both IaC and cloud accounts, you can prevent misconfigured modules from being provisioned and identify errors introduced manually. Bridgecrew also takes policy-as-code a step further, transforming runtime errors into Terraform fixes.

Terraform is a powerful tool for managing your infrastructure. As your infrastructure grows and Terraform configurations multiply, it's important to secure that infrastructure from development to production. Learning these and other best practices will help you get the most out of Terraform.

Ready to get even more out of your Terraform environment and help with your security and compliance policies? Try out our Terraform workshop, where you'll:

- Get an overview of DevSecOps and Terraform infrastructure as code (IaC)
- Scan IaC files for misconfigurations locally
- Set up CI/CD pipelines to automate security scanning and policy enforcement
- Fix IaC security errors and AWS resource misconfigurations with Bridgecrew

Or maybe you're ready to try it for yourself. If so, schedule a demo of Bridgecrew or get started for free today.