Chapter 6. Managing Secrets with Terraform

At some point, you and your software will be entrusted with a variety of secrets, such as database passwords, API keys, TLS certificates, SSH keys, GPG keys, and so on. This is all sensitive data that, if it were to get into the wrong hands, could do a lot of damage to your company and its customers. If you build software, it is your responsibility to keep those secrets secure.

For example, consider the following Terraform code for deploying a database:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # How to set these parameters securely?
  username = "???"
  password = "???"
}
```

This code requires you to set two secrets, the username and password, which are the credentials for the master user of the database. If the wrong person gets access to them, it could be catastrophic, as these credentials give you superuser access to that database and all the data within it. So, how do you keep these secrets secure?

This is part of the broader topic of *secrets management*, which is the focus of this chapter. This chapter will cover:

- Secret management basics
- Secret management tools
- Secret management tools with Terraform

Secret Management Basics

The first rule of secrets management is:

> *Do not store secrets in plain text.*

The second rule of secrets management is:

> *DO NOT STORE SECRETS IN PLAIN TEXT.*

Seriously, don't do it. For example, do *not* hardcode your database credentials directly in your Terraform code and check it into version control:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # DO NOT DO THIS!!!
  username = "admin"
  password = "password"
  # DO NOT DO THIS!!!
}
```

Storing secrets in plain text in version control is a *bad idea*. Here are just a few of the reasons why:

Anyone who has access to the version control system has access to that secret.

> In the preceding example, every single developer at your company who can access that Terraform code will have access to the master credentials for your database.

Every computer that has access to the version control system keeps a copy of that secret.

> Every single computer that has ever checked out that repo may still have a copy of that secret on its local hard drive. That includes the computer of every developer on your team, every computer involved in CI (e.g., Jenkins, CircleCI, GitLab, etc.), every computer in-

volved in version control (e.g., GitHub, GitLab, BitBucket), every computer involved in deployment (e.g., all your pre-prod and prod environments), every computer involved in backup (e.g., CrashPlan, Time Machine, etc.), and so on.

Every piece of software you run has access to that secret.

Because the secrets are sitting in plain text on so many hard drives, every single piece of software running on any of those computers can potentially read that secret.

There's no way to audit or revoke access to that secret.

When secrets are sitting on hundreds of hard drives in plain text, you have no way to know who accessed them (there's no audit log) and no easy way to revoke access.

In short, if you store secrets in plain text, you are giving malicious actors (e.g., hackers, competitors, disgruntled former employees) countless ways to access your company's most sensitive data—e.g., by compromising the version control system, or by compromising any of the computers you use, or by compromising any piece of software on any of those computers —and you'll have no idea if you were compromised or have any easy way to fix things if you were.

Therefore, it's essential that you use a proper *secret management tool* to store your secrets.

Secret Management Tools

A comprehensive overview of all aspects of secret management is beyond the scope of this book, but to be able to use secret management tools with Terraform, it's worth briefly touching on the following topics:

- The types of secrets you store
- The way you store secrets
- The interface you use to access secrets
- A comparison of secret management tools

The Types of Secrets You Store

There are three primary types of secrets: personal secrets, customer secrets, and infrastructure secrets.

Personal secrets

Belong to an individual. Examples include the usernames and passwords for websites you visit, your SSH keys, and your Pretty Good Privacy (PGP) keys.

Customer secrets

Belong to your customers. Note that if you run software for other employees of your company—e.g., you manage your company's internal Active Directory server—then those other employees are your customers. Examples include the usernames and passwords that your customers use to log into your product, personally identifiable info (PII) for your customers, and personal health information (PHI) for your customers.

Infrastructure secrets

Belong to your infrastructure. Examples include database passwords, API keys, and TLS certificates.

Most secret management tools are designed to store exactly one of these types of secrets, and while you could try to force it to store the other types, that's rarely a good idea from a security or usability standpoint. For example, the way you store passwords that are infrastructure secrets is completely different from how you store passwords that are customer secrets: for the former, you'd typically use an encryption algorithm such as AES (Advanced Encryption Standard), perhaps with a nonce, as you need to be able to decrypt the secrets and get back the original password; on the other hand, for the latter, you'd typically use a hashing algorithm (e.g., bcrypt) with a salt, as there should be no way to get back the original password. Using the wrong approach can be catastrophic, so use the right tool for the job!

The Way You Store Secrets

The two most common strategies for storing secrets are to use either a file-based secret store or a centralized secret store.

File-based secret stores store secrets in encrypted files, which are typically checked into version control. To encrypt the files, you need an encryption key. This key is itself a secret! This creates a bit of a conundrum: How do you securely store that key? You can't check the key into version control as plain text, as then there’s no point of encrypting anything with it. You could encrypt the key with another key, but then all you’ve done is kicked the can down the road, as you still have to figure out how to securely store that second key.

The most common solution to this conundrum is to store the key in a *key management service* (KMS) provided by your cloud provider, such as AWS KMS, GCP KMS, or Azure Key Vault. This solves the kick-the-can-down-the-road problem by trusting the cloud provider to securely store the secret and manage access to it. Another option is to use PGP keys. Each developer can have their own PGP key, which consists of a *public key* and a *private key*. If you encrypt a secret with one or more public keys, only developers with the corresponding private keys will be able to decrypt those secrets. The private keys, in turn, are protected by a password that the developer either memorizes or stores in a personal secrets manager.

Centralized secret stores are typically web services that you talk to over the network that encrypt your secrets and store them in a data store such as MySQL, PostgreSQL, DynamoDB, etc. To encrypt these secrets, these centralized secret stores need an encryption key. Typically, the encryption key is managed by the service itself, or the service relies on a cloud provider’s KMS.

The Interface You Use to Access Secrets

Most secret management tools can be accessed via an API, CLI, and/or UI.

Just about all centralized secret stores expose an API that you can consume via network requests: e.g., a REST API you access over HTTP. The API is convenient for when your code needs to programmatically read secrets. For example, when an app is booting up, it can make an API call to your centralized secret store to retrieve a database password. Also, as you’ll see later in this chapter, you can write Terraform code that, under the hood, uses a centralized secret store’s API to retrieve secrets.

All the file-based secret stores work via a *command-line interface (CLI)*. Many of the centralized secret stores also provide CLI tools that, under the hood, make API calls to the service. CLI tools are a convenient way for developers to access secrets (e.g., using a few CLI commands to encrypt a file) and for scripting (e.g., writing a script to encrypt secrets).

Some of the centralized secret stores also expose a *user interface (UI)* via the web, desktop, or mobile. This is potentially an even more convenient way for everyone on your team to access secrets.

A Comparison of Secret Management Tools

Table 6-1 shows a comparison of popular secret management tools, broken down by the three considerations defined in the previous sections.

Table 6-1. A comparison of secret management tools

	Types of secrets	Secret storage	Secret interface
HashiCorp Vault	Infrastructure[a]	Centralized service	UI, API, CLI
AWS Secrets Manager	Infrastructure	Centralized service	UI, API, CLI
Google Secrets Manager	Infrastructure	Centralized service	UI, API, CLI
Azure Key Vault	Infrastructure	Centralized service	UI, API, CLI
Confidant	Infrastructure	Centralized service	UI, API, CLI
Keywhiz	Infrastructure	Centralized service	API, CLI
sops	Infrastructure	Files	CLI
git-secret	Infrastructure	Files	CLI
1Password	Personal	Centralized service	UI, API, CLI
LastPass	Personal	Centralized service	UI, API, CLI
Bitwarden	Personal	Centralized service	UI, API, CLI
KeePass	Personal	Files	UI, CLI
Keychain (macOS)	Personal	Files	UI, CLI

	Types of secrets	**Secret storage**	**Secret interface**
Credential Manager (Windows)	Personal	Files	UI, CLI
pass	Personal	Files	CLI
Active Directory	Customer	Centralized service	UI, API, CLI
Auth0	Customer	Centralized service	UI, API, CLI
Okta	Customer	Centralized service	UI, API, CLI
OneLogin	Customer	Centralized service	UI, API, CLI
Ping	Customer	Centralized service	UI, API, CLI
AWS Cognito	Customer	Centralized service	UI, API, CLI

[a] Vault supports multiple *secret engines*, most of which are designed for infrastructure secrets, but a few support customer secrets as well.

Since this is a book about Terraform, from here on out, I'll mostly be focusing on secret management tools designed for infrastructure secrets that are accessed through an API or the CLI (although I'll mention personal secret management tools from time to time too, as those often contain the secrets you need to authenticate to the infrastructure secret tools).

Secret Management Tools with

Terraform

Let's now turn to how to use these secret management tools with Terraform, going through each of the three places where your Terraform code is likely to brush up against secrets:

- Providers
- Resources and data sources
- State files and plan files

Providers

Typically, your first exposure to secrets when working with Terraform is when you have to authenticate to a provider. For example, if you want to run `terraform apply` on code that uses the AWS Provider, you'll need to first authenticate to AWS, and that typically means using your access keys, which are secrets. How should you store those secrets? And how should you make them available to Terraform?

There are many ways to answer these questions. One way you should *not* do it, even though it occasionally comes up in the Terraform documentation, is by putting secrets directly into the code, in plain text:

```
provider "aws" {
  region = "us-east-2"

  # DO NOT DO THIS!!!
  access_key = "(ACCESS_KEY)"
  secret_key = "(SECRET_KEY)"
  # DO NOT DO THIS!!!
}
```

Storing credentials this way, in plain text, is *not* secure, as discussed earlier in this chapter. Moreover, it's also not practical, as this would hardcode you to using one set of credentials for all users of this module, whereas in most cases, you'll need different credentials on different computers (e.g., when different developers or your CI server runs `apply`) and in different environments (dev, stage, prod).

There are several techniques that are far more secure for storing your credentials and making them accessible to Terraform providers. Let's take a look at these techniques, grouping them based on the user who is running Terraform:

Human users

Developers running Terraform on their own computers.

Machine users

Automated systems (e.g., a CI server) running Terraform with no humans present.

Human users

Just about all Terraform providers allow you to specify your credentials in some way other than putting them directly into the code. The most common option is to use environment variables. For example, here's how you use environment variables to authenticate to AWS:

```
$  export AWS_ACCESS_KEY_ID=(YOUR_ACCESS_KEY_ID)
$  export AWS_SECRET_ACCESS_KEY=(YOUR_SECRET_ACCESS_KEY)
```

Setting your credentials as environment variables keeps plain-text secrets out of your code, ensures that everyone running Terraform has to provide their own credentials, and ensures that credentials are only ever stored in memory, and not on disk.[1]

One important question you may ask is where to store the access key ID and secret access key in the first place. They are too long and random to memorize, but if you store them on your computer in plain text, then you're still putting those secrets at risk. Since this section is focused on human users, the solution is to store your access keys (and other secrets) in a secret manager designed for personal secrets. For example, you could store your access keys in 1Password or LastPass and copy/paste them into the `export` commands in your terminal.

If you're using these credentials frequently on the CLI, an even more convenient option is to use a secret manager that supports a CLI interface. For example, 1Password offers a CLI tool called `op`. On Mac and Linux, you can use `op` to authenticate to 1Password on the CLI as follows:

```
$ eval $(op signin my)
```

Once you've authenticated, assuming you had used the 1Password app to store your access keys under the name "aws-dev" with fields "id" and "secret", here's how you can use `op` to set those access keys as environment variables:

```
$ export AWS_ACCESS_KEY_ID=$(op get item 'aws-dev' --fields 'id')
$ export AWS_SECRET_ACCESS_KEY=$(op get item 'aws-dev' --fields 'secret'
```

While tools like 1Password and `op` are great for general-purpose secrets management, for certain providers, there are dedicated CLI tools to make this even easier. For example, for authenticating to AWS, you can use the open source tool `aws-vault`. You can save your access keys using the `aws-vault add` command under a *profile* named `dev` as follows:

```
$ aws-vault add dev
Enter Access Key Id: (YOUR_ACCESS_KEY_ID)
Enter Secret Key: (YOUR_SECRET_ACCESS_KEY)
```

Under the hood, `aws-vault` will store these credentials securely in your operating system's native password manager (e.g., Keychain on macOS, Credential Manager on Windows). Once you've stored these credentials, now you can authenticate to AWS for any CLI command as follows:

```
$ aws-vault exec <PROFILE> -- <COMMAND>
```

where `PROFILE` is the name of a profile you created earlier via the `add` command (e.g., `dev`) and `COMMAND` is the command to execute. For example, here's how you can use the `dev` credentials you saved earlier to run `terraform apply`:

```
$ aws-vault exec dev -- terraform apply
```

The `exec` command automatically uses AWS STS to fetch temporary credentials and exposes them as environment variables to the command you're executing (in this case, `terraform apply`). This way, not only are your permanent credentials stored in a secure manner (in your operating system's native password manager), but now, you're also only exposing temporary credentials to any process you run, so the risk of leaking credentials is minimized. `aws-vault` also has native support for assuming

IAM roles, using multifactor authentication (MFA), logging into accounts on the web console, and more.

Machine users

Whereas a human user can rely on a memorized password, what do you do in cases where there's no human present? For example, if you're setting up a continuous integration / continuous delivery (CI/CD) pipeline to automatically run Terraform code, how do you securely authenticate that pipeline? In this case, you are dealing with authentication for a *machine user*. The question is, how do you get one machine (e.g., your CI server) to authenticate itself to another machine (e.g., AWS API servers) without storing any secrets in plain text?

The solution here heavily depends on the type of machines involved: that is, the machine you're authenticating *from* and the machine you're authenticating *to*. Let's go through three examples:

- CircleCI as a CI server, with stored secrets
- EC2 Instance running Jenkins as a CI server, with IAM roles
- GitHub Actions as a CI server, with OIDC

WARNING: SIMPLIFIED EXAMPLES

This section contains examples that fully flush out how to handle provider authentication in a CI/CD context, but all other aspects of the CI/CD workflow are highly simplified. You'll see more complete, end-to-end, production-ready CI/CD workflows in Chapter 9.

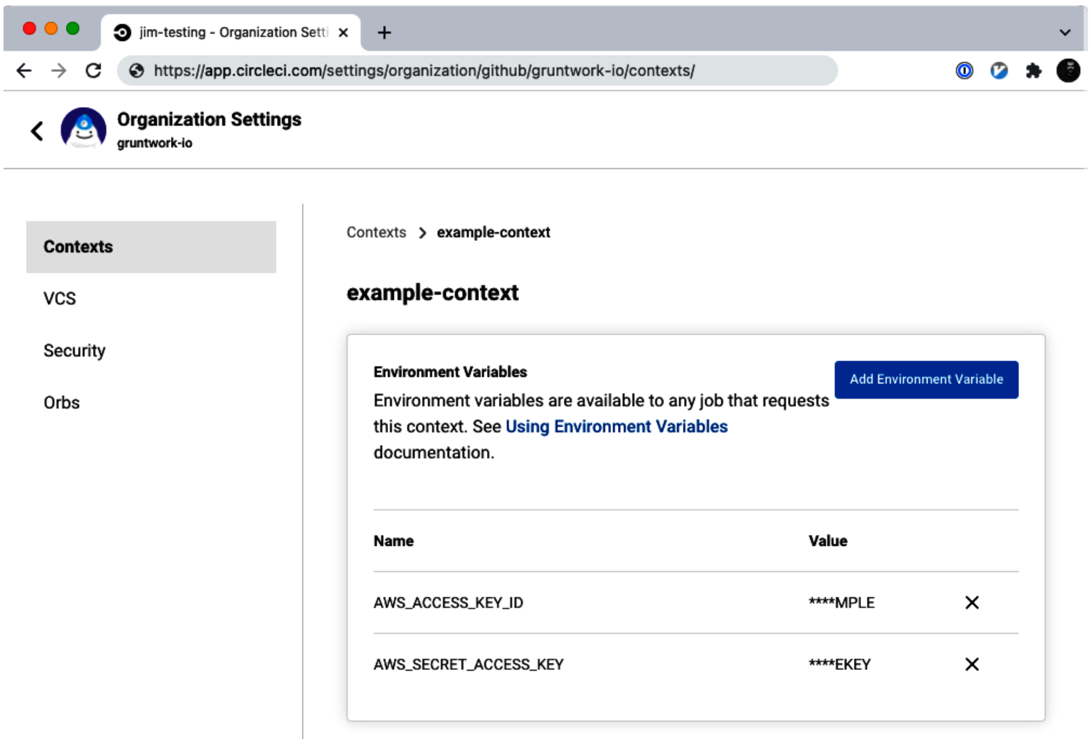
CircleCI as a CI server, with stored secrets

Let's imagine that you want to use CircleCI, a popular managed CI/CD platform, to run Terraform code. With CircleCI, you configure your build steps in a *.circleci/config.yml* file, where you might define a job to run `terraform apply` that looks like this:

```
version: '2.1'
orbs:
  # Install Terraform using a CircleCi Orb
  terraform: circleci/terraform@1.1.0
jobs:
  # Define a job to run 'terraform apply'
```

```
    terraform_apply:
      executor: terraform/default
      steps:
        - checkout            # git clone the code
        - terraform/init     # Run 'terraform init'
        - terraform/apply   # Run 'terraform apply'
  workflows:
    # Create a workflow to run the 'terraform apply' job defined above
    deploy:
      jobs:
        - terraform_apply
      # Only run this workflow on commits to the main branch
      filters:
        branches:
          only:
            - main
```

With a tool like CircleCI, the way to authenticate to a provider is to create a machine user in that provider (that is, a user solely used for automation, and not by any human), store the credentials for that machine user in CircleCI in what's called a *CircleCI Context*, and when your build runs, CircleCI will expose the credentials in that Context to your workflows as environment variables. For example, if your Terraform code needs to authenticate to AWS, you would create a new IAM user in AWS, give that IAM user the permissions it needs to deploy your Terraform changes, and manually copy that IAM user's access keys into a CircleCI Context, as shown in Figure 6-1.


jim-testing - Organization Setti
https://app.circleci.com/settings/organization/github/gruntwork-io/contexts/
Organization Settings
gruntwork-io
Contexts
VCS
Security
Orbs
Contexts > example-context
Add Environment Variable

example-context

Environment Variables
Environment variables are available to any job that requests this context. See Using Environment Variables documentation.

Name	Value
AWS_ACCESS_KEY_ID	****MPLE
AWS_SECRET_ACCESS_KEY	****EKEY

Figure 6-1. A CircleCI Context with AWS credentials.

Finally, you update the `workflows` in your *.circleci/config.yml* file to use your CircleCI Context via the `context` parameter:

```
workflows:
  # Create a workflow to run the 'terraform apply' job defined above
  deploy:
    jobs:
      - terraform_apply
    # Only run this workflow on commits to the main branch
    filters:
      branches:
        only:
          - main
    # Expose secrets in the CircleCI context as environment variables
    context:
      - example-context
```

When your build runs, CircleCI will automatically expose the secrets in that Context as environment variables—in this case, as the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`—and `terraform apply` will automatically use those environment variables to authenticate to your provider.

The biggest drawbacks to this approach are that (a) you have to manually manage credentials, and (b) as a result, you have to use permanent credentials, which once saved in CircleCI, rarely (if ever) change. The next two examples show alternative approaches.

EC2 Instance running Jenkins as a CI server, with IAM roles

If you're using an EC2 Instance to run Terraform code—e.g., you're running Jenkins on an EC2 Instance as a CI server—the solution I recommend for machine user authentication is to give that EC2 Instance an IAM role. An *IAM role* is similar to an IAM user, in that it's an entity in AWS that can be granted IAM permissions. However, unlike IAM users, IAM roles are not associated with any one person and do not have permanent credentials (password or access keys). Instead, a role can be *assumed* by other IAM entities: for example, an IAM user might assume a role to temporarily get access to different permissions than they normally have; many AWS services, such as EC2 Instances, can assume IAM roles to grant those services permissions in your AWS account.

For example, here's code you've seen many times to deploy an EC2 Instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

To create an IAM role, you must first define an *assume role policy*, which is an IAM Policy that defines who is allowed to assume the IAM role. You could write the IAM Policy in raw JSON, but Terraform has a convenient `aws_iam_policy_document` data source that can create the JSON for you. Here's how you can use an `aws_iam_policy_document` to define an assume role policy that allows the EC2 service to assume an IAM role:

```
data "aws_iam_policy_document" "assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]

    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}
```

Now, you can use the `aws_iam_role` resource to create an IAM role and pass it the JSON from your `aws_iam_policy_document` to use as the assume role policy:

```
resource "aws_iam_role" "instance" {
  name_prefix        = var.name
  assume_role_policy = data.aws_iam_policy_document.assume_role.json
}
```

You now have an IAM role, but by default, IAM roles don't give you any permissions. So, the next step is to attach one or more IAM policies to the IAM role that specify what you can actually do with the role once you've assumed it. Let's imagine that you're using Jenkins to run Terraform code that deploys EC2 Instances. You can use the `aws_iam_policy_document` data source to define an IAM Policy that gives admin permissions over EC2 Instances as follows:

```
data "aws_iam_policy_document" "ec2_admin_permissions" {
  statement {
    effect    = "Allow"
    actions   = ["ec2:*"]
    resources = ["*"]
  }
}
```

And you can attach this policy to your IAM role using the `aws_iam_role_policy` resource:

```
resource "aws_iam_role_policy" "example" {
  role   = aws_iam_role.instance.id
  policy = data.aws_iam_policy_document.ec2_admin_permissions.json
}
```

The final step is to allow your EC2 Instance to automatically assume that IAM role by creating an *instance profile*:

```
resource "aws_iam_instance_profile" "instance" {
  role = aws_iam_role.instance.name
}
```

And then tell your EC2 Instance to use that instance profile via the `iam_instance_profile` parameter:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

  # Attach the instance profile
  iam_instance_profile = aws_iam_instance_profile.instance.name
}
```

Under the hood, AWS runs an *instance metadata endpoint* on every EC2 Instance at *http://169.254.169.254*. This is an endpoint that can only be reached by processes running on the instance itself, and those processes can use that endpoint to fetch metadata about the instance. For example, if you SSH to an EC2 Instance, you can query this endpoint using `curl`:

```
$ ssh ubuntu@<IP_OF_INSTANCE>
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-1022-aws x86_64)
(...)

$ curl http://169.254.169.254/latest/meta-data/
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
events/
hibernation/
hostname
identity-credentials/
(...)
```

If the instance has an IAM role attached (via an instance profile), that metadata will include AWS credentials that can be used to authenticate to AWS and assume that IAM role. Any tool that uses the AWS SDK, such as Terraform, knows how to use these instance metadata endpoint credentials automatically, so as soon as you run `terraform apply` on the EC2 Instance with this IAM role, your Terraform code will authenticate as this IAM role, which will thereby grant your code the EC2 admin permissions it needs to run successfully.[2]

For any automated process running in AWS, such as a CI server, IAM roles provide a way to authenticate (a) without having to manage credentials manually, and (b) the credentials AWS provides via the instance metadata endpoint are always temporary, and rotated automatically. These are two big advantages over the manually managed, permanent credentials with a tool like CircleCI that runs outside of your AWS account. However, as you'll see in the next example, in some cases, it's possible to have these same advantages for external tools, too.

GitHub Actions as a CI server, with OIDC

GitHub Actions is another popular managed CI/CD platform you might want to use to run Terraform. In the past, GitHub Actions required you to manually copy credentials around, just like CircleCI. However, as of 2021, GitHub Actions offers a better alternative: *Open ID Connect (OIDC)*. Using OIDC, you can establish a trusted link between the CI system and your cloud provider (GitHub Actions supports AWS, Azure, and Google Cloud) so that your CI system can authenticate to those providers without having to manage any credentials manually.

You define GitHub Actions workflows in YAML files in a *.github/workflows* folder, such as the *terraform.yml* file shown here:

```
name: Terraform Apply
# Only run this workflow on commits to the main branch
on:
  push:
    branches:
      - 'main'
jobs:
  TerraformApply:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      # Run Terraform using HashiCorp's setup-terraform Action
      - uses: hashicorp/setup-terraform@v1
          with:
            terraform_version: 1.1.0
            terraform_wrapper: false
        run: |
          terraform init
          terraform apply -auto-approve
```

If your Terraform code talks to a provider such as AWS, you need to provide a way for this workflow to authenticate to that provider. To do this using OIDC,[3] the first step is to create an *IAM OIDC identity provider* in your AWS account, using the `aws_iam_openid_connect_provider` resource, and to configure it to trust the GitHub Actions thumbprint, fetched via the `tls_certificate` data source:

```
# Create an IAM OIDC identity provider that trusts GitHub
resource "aws_iam_openid_connect_provider" "github_actions" {
  url             = "https://token.actions.githubusercontent.com"
  client_id_list  = ["sts.amazonaws.com"]
  thumbprint_list = [
    data.tls_certificate.github.certificates[0].sha1_fingerprint
  ]
}

# Fetch GitHub's OIDC thumbprint
data "tls_certificate" "github" {
  url = "https://token.actions.githubusercontent.com"
}
```

Now, you can create IAM roles exactly as in the previous section—e.g., an IAM role with EC2 admin permissions attached—except the assume role policy for those IAM roles will look different:

```
data "aws_iam_policy_document" "assume_role_policy" {
  statement {
    actions = ["sts:AssumeRoleWithWebIdentity"]
    effect  = "Allow"

    principals {
      identifiers = [aws_iam_openid_connect_provider.github_actions.arn
      type        = "Federated"
    }

    condition {
      test     = "StringEquals"
      variable = "token.actions.githubusercontent.com:sub"
      # The repos and branches defined in var.allowed_repos_branches
      # will be able to assume this IAM role
      values = [
        for a in var.allowed_repos_branches :
        "repo:${a["org"]}/${a["repo"]}:ref:refs/heads/${a["branch"]}"
      ]
    }
  }
}
```

This policy allows the IAM OIDC identity provider to assume the IAM role via federated authentication. Note the `condition` block, which ensures that only the specific GitHub repos and branches you specify via the `allowed_repos_branches` input variable can assume this IAM role:

```
variable "allowed_repos_branches" {
  description = "GitHub repos/branches allowed to assume the IAM role."
  type = list(object({
    org    = string
    repo   = string
    branch = string
  }))
  # Example:
  # allowed_repos_branches = [
  #   {
```

```
  #     org    = "brikis98"
  #     repo   = "terraform-up-and-running-code"
  #     branch = "main"
  #   }
  # ]
}
```

This is important to ensure you don't accidentally allow *all* GitHub repos to authenticate to your AWS account! You can now configure your builds in GitHub Actions to assume this IAM role. First, at the top of your workflow, give your build the `id-token: write` permission:

```
permissions:
  id-token: write
```

Next, add a build step just before running Terraform to authenticate to AWS using the `configure-aws-credentials` action:

```
      # Authenticate to AWS using OIDC
      - uses: aws-actions/configure-aws-credentials@v1
        with:
          # Specify the IAM role to assume here
          role-to-assume: arn:aws:iam::123456789012:role/example-role
          aws-region: us-east-2

      # Run Terraform using HashiCorp's setup-terraform Action
      - uses: hashicorp/setup-terraform@v1
          with:
            terraform_version: 1.1.0
            terraform_wrapper: false
        run: |
          terraform init
          terraform apply -auto-approve
```

Now, when you run this build in one of the repos and branches in the `allowed_repos_branches` variable, GitHub will be able to assume your IAM role automatically, using temporary credentials, and Terraform will authenticate to AWS using that IAM role, all without having to manage any credentials manually.

Resources and Data Sources

The next place you'll run into secrets with your Terraform code is with resources and data sources. For example, you saw earlier in the chapter the example of passing database credentials to the `aws_db_instance` resource:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # DO NOT DO THIS!!!
  username = "admin"
  password = "password"
  # DO NOT DO THIS!!!
}
```

I've said it multiple times in this chapter already, but it's such an important point that it's worth repeating again: storing those credentials in the code, as plain text, is a bad idea. So, what's a better way to do it?

There are three main techniques you can use:

- Environment variables
- Encrypted files
- Secret stores

Environment variables

This first technique, which you saw back in Chapter 3, as well as earlier in this chapter when talking about providers, keeps plain-text secrets out of your code by taking advantage of Terraform's native support for reading environment variables.

To use this technique, declare variables for the secrets you wish to pass in:

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}

variable "db_password" {
  description = "The password for the database"
  type        = string
  sensitive   = true
}
```

Just as in Chapter 3, these variables are marked with `sensitive = true` to indicate they contain secrets (so Terraform won't log the values when you run `plan` or `apply`), and these variables do not have a `default` (so as not to store secrets in plain text). Next, pass the variables to the Terraform resources that need those secrets:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # Pass the secrets to the resource
  username = var.db_username
  password = var.db_password
}
```

Now you can pass in a value for each variable `foo` by setting the environment variable `TF_VAR_foo`:

```
$  export TF_VAR_db_username=(DB_USERNAME)
$  export TF_VAR_db_password=(DB_PASSWORD)
```

Passing in secrets via environment variables helps you avoid storing secrets in plain text in your code, but it doesn't answer an important question: How do you store the secrets securely? One nice thing about using environment variables is that they work with almost any type of secrets management solution. For example, one option is to store the secrets in a personal secrets manager (e.g., 1Password) and manually set those secrets as environment variables in your terminal. Another option is to store the secrets in a centralized secret store (e.g., HashiCorp Vault) and write a script that uses that secret store's API or CLI to read those secrets out and set them as environment variables.

Using environment variables has the following advantages:

- Keep plain-text secrets out of your code and version control system.
- Storing secrets is easy, as you can use just about any other secret management solution. That is, if your company already has a way to manage secrets, you can typically find a way to make it work with environment variables.
- Retrieving secrets is easy, as reading environment variables is straightforward in every language.
- Integrating with automated tests is easy, as you can easily set the environment variables to mock values.
- Using environment variables doesn't cost any money, unlike some of the other secret management solutions discussed later.

Using environment variables has the following drawbacks:

- Not everything is defined in the Terraform code itself. This makes understanding and maintaining the code harder. Everyone using your code has to know to take extra steps to either manually set these environment variables or run a wrapper script.
- Standardizing secret management practices is harder. Since all the management of secrets happens outside of Terraform, the code doesn't enforce any security properties, and it's possible someone is still managing the secrets in an insecure way (e.g., storing them in plain text).
- Since the secrets are not versioned, packaged, and tested with your code, configuration errors are more likely, such as adding a new secret in one environment (e.g., staging) but forgetting to add it in another environment (e.g., production).

Encrypted files

The second technique relies on encrypting the secrets, storing the ciphertext in a file, and checking that file into version control.

To encrypt some data, such as some secrets in a file, you need an encryption key. As mentioned earlier in this chapter, this encryption key is itself a secret, so you need a secure way to store it. The typical solution is to either use your cloud provider's KMS (e.g., AWS KMS, Google KMS, Azure

Key Vault) or to use the PGP keys of one or more developers on your team.

Let's look at an example that uses AWS KMS. First, you'll need to create a KMS *Customer Managed Key* (CMK), which is an encryption key that AWS manages for you. To create a CMK, you first have to define a *key policy*, which is an IAM Policy that defines who can use that CMK. To keep this example simple, let's create a key policy that gives the current user admin permissions over the CMK. You can fetch the current user's information—their username, ARN, etc.—using the `aws_caller_identity` data source:

```
provider "aws" {
  region = "us-east-2"
}

data "aws_caller_identity" "self" {}
```

And now you can use the `aws_caller_identity` data source's outputs inside an `aws_iam_policy_document` data source to create a key policy that gives the current user admin permissions over the CMK:

```
data "aws_iam_policy_document" "cmk_admin_policy" {
  statement {
    effect    = "Allow"
    resources = ["*"]
    actions   = ["kms:*"]
    principals {
      type        = "AWS"
      identifiers = [data.aws_caller_identity.self.arn]
    }
  }
}
```

Next, you can create the CMK using the `aws_kms_key` resource:

```
resource "aws_kms_key" "cmk" {
  policy = data.aws_iam_policy_document.cmk_admin_policy.json
}
```

Note that, by default, KMS CMKs are only identified by a long numeric identifier (e.g., `b7670b0e-ed67-28e4-9b15-0d61e1485be3`), so it's a good practice to also create a human-friendly *alias* for your CMK using the `aws_kms_alias` resource:

```
resource "aws_kms_alias" "cmk" {
  name          = "alias/kms-cmk-example"
  target_key_id = aws_kms_key.cmk.id
}
```

The preceding alias will allow you to refer to your CMK as `alias/kms-cmk-example` when using the AWS API and CLI, rather than a long identifier such as `b7670b0e-ed67-28e4-9b15-0d61e1485be3`. Once you've created the CMK, you can start using it to encrypt and decrypt data. Note that, by design, you'll never be able to see (and, therefore, to accidentally leak) the underlying encryption key. Only AWS has access to that encryption key, but you can make use of it by using the AWS API and CLI, as described next.

First, create a file called *db-creds.yml* with some secrets in it, such as the database credentials:

```
username: admin
password: password
```

Note: do *not* check this file into version control, as you haven't encrypted it yet! To encrypt this data, you can use the `aws kms encrypt` command and write the resulting ciphertext to a new file. Here's a small Bash script (for Linux/Unix/macOS) called *encrypt.sh* that performs these steps using the AWS CLI:

```
CMK_ID="$1"
AWS_REGION="$2"
INPUT_FILE="$3"
OUTPUT_FILE="$4"

echo "Encrypting contents of $INPUT_FILE using CMK $CMK_ID..."
ciphertext=$(aws kms encrypt \
  --key-id "$CMK_ID" \
  --region "$AWS_REGION" \
  --plaintext "fileb://$INPUT_FILE" \
  --output text \
  --query CiphertextBlob)
```

```
echo "Writing result to $OUTPUT_FILE..."
echo "$ciphertext" > "$OUTPUT_FILE"

echo "Done!"
```

Here’s how you can use *encrypt.sh* to encrypt the *db-creds.yml* file with the KMS CMK you created earlier and store the resulting ciphertext in a new file called *db-creds.yml.encrypted*:

```
$ ./encrypt.sh \
  alias/kms-cmk-example \
  us-east-2 \
  db-creds.yml \
  db-creds.yml.encrypted

Encrypting contents of db-creds.yml using CMK alias/kms-cmk-example...
Writing result to db-creds.yml.encrypted...
Done!
```

You can now delete *db-creds.yml* (the plain-text file) and safely check *db-creds.yml.encrypted* (the encrypted file) into version control. At this point, you have an encrypted file with some secrets inside of it, but how do you make use of that file in your Terraform code?

The first step is to decrypt the secrets in this file using the `aws_kms_secrets` data source:

```
data "aws_kms_secrets" "creds" {
  secret {
    name    = "db"
    payload = file("${path.module}/db-creds.yml.encrypted")
  }
}
```

The preceding code reads *db-creds.yml.encrypted* from disk using the `file` helper function and, assuming you have permissions to access the corresponding key in KMS, decrypts the contents. That gives you back the contents of the original *db-creds.yml* file, so the next step is to parse the YAML as follows:

```
locals {
  db_creds = yamldecode(data.aws_kms_secrets.creds.plaintext["db"])
}
```

This code pulls out the database secrets from the `aws_kms_secrets` data source, parses the YAML, and stores the results in a local variable called `db_creds`. Finally, you can read the username and password from `db_creds` and pass those credentials to the `aws_db_instance` resource:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # Pass the secrets to the resource
  username = local.db_creds.username
  password = local.db_creds.password
}
```

So now you have a way to store secrets in an encrypted file, which are safe to check into version control, and you have a way to read those secrets back out of the file in your Terraform code automatically.

One thing to note with this approach is that working with encrypted files is awkward. To make a change, you have to locally decrypt the file with a long `aws kms decrypt` command, make some edits, re-encrypt the file with another long `aws kms encrypt` command, and the whole time, be extremely careful to not accidentally check the plain-text data into version control or leave it sitting behind forever on your computer. This is a tedious and error-prone process.

One way to make this less awkward is to use an open source tool called sops. When you run `sops <FILE>`, sops will automatically decrypt `FILE` and open your default text editor with the plain-text contents. When you're done editing and exit the text editor, sops will automatically encrypt the contents. This way, the encryption and decryption are mostly transparent, with no need to run long `aws kms` commands and less chance of accidentally checking plain-text secrets into version control. As of 2022, sops can work with files encrypted via AWS KMS, GCP KMS, Azure Key Vault, or PGP keys. Note that Terraform doesn't yet have native support for decrypting files that were encrypted by sops, so you'll either need to use a third-party provider such as carlpett/sops or, if you're a Terragrunt user, you can use the built-in `sops_decrypt_file` function.

Using encrypted files has the following advantages:

- Keep plain-text secrets out of your code and version control system.
- Your secrets are stored in an encrypted format in version control, so they are versioned, packaged, and tested with the rest of your code. This helps reduce configuration errors, such as adding a new secret in one environment (e.g., staging) but forgetting to add it in another environment (e.g., production).
- Retrieving secrets is easy, assuming the encryption format you're using is natively supported by Terraform or a third-party plugin.
- It works with a variety of different encryption options: AWS KMS, GCP KMS, PGP, etc.
- Everything is defined in the code. There are no extra manual steps or wrapper scripts required (although sops integration does require a third-party plugin).

Using encrypted files has the following drawbacks:

- Storing secrets is harder. You either have to run lots of commands (e.g., `aws kms encrypt`) or use an external tool such as sops. There's a learning curve to using these tools correctly and securely.
- Integrating with automated tests is harder, as you will need to do extra work to make encryption keys and encrypted test data available for your test environments.
- The secrets are now encrypted, but as they are still stored in version control, rotating and revoking secrets is hard. If anyone ever compromises the encryption key, they can go back and decrypt all the secrets that were ever encrypted with it.
- The ability to audit who accessed secrets is minimal. If you're using a cloud key management service (e.g., AWS KMS), it will likely maintain an audit log of who used an encryption key, but you won't be able to tell what the key was actually used for (i.e., what secrets were accessed).
- Most managed key services cost a small amount of money. For example, each key you store in AWS KMS costs $1/month, plus $0.03 per 10,000 API calls, where each decryption and encryption operation requires one API call. A typical usage pattern, where you have a small

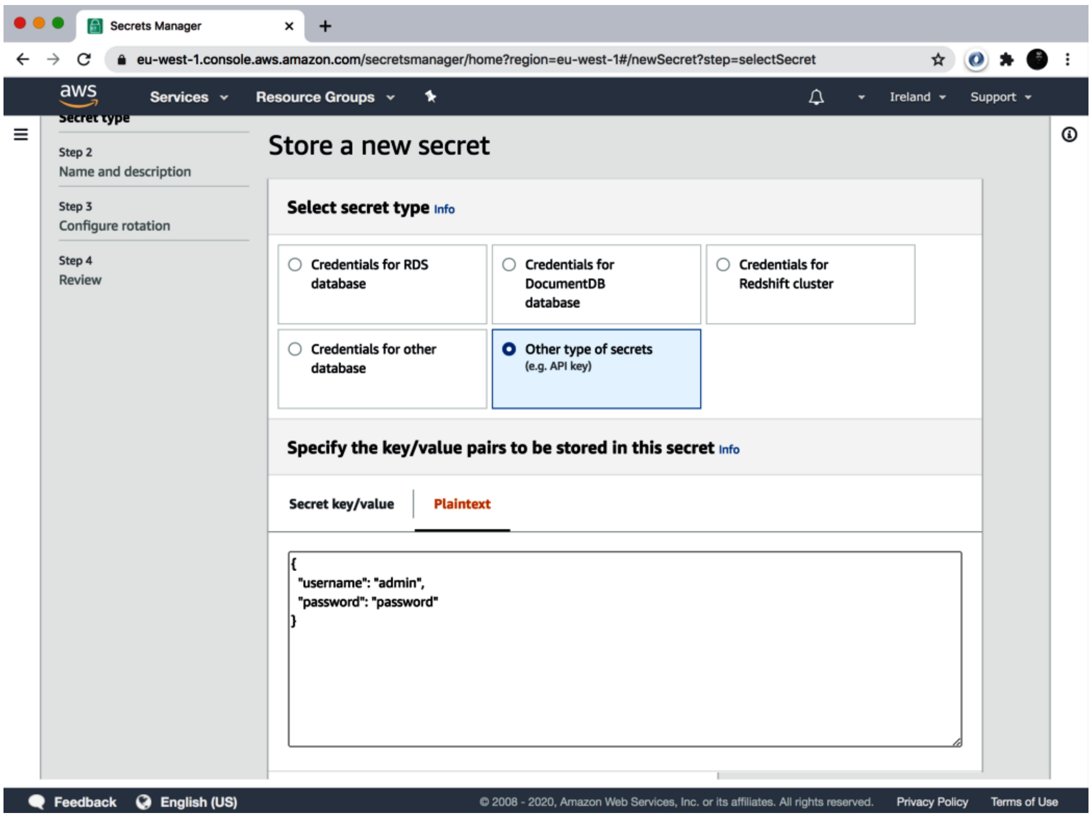
number of keys in KMS and your apps use those keys to decrypt secrets during boot, usually costs $1–$10/month. For larger deployments, where you have dozens of apps and hundreds of secrets, the price is typically in the $10–$50/month range.

- Standardizing secret management practices is harder. Different developers or teams may use different ways to store encryption keys or manage encrypted files, and mistakes are relatively common, such as not using encryption correctly or accidentally checking in a plaintext file into version control.

Secret stores

The third technique relies on storing your secrets in a centralized secret store.

Some of the more popular secret stores are AWS Secrets Manager, Google Secret Manager, Azure Key Vault, and HashiCorp Vault. Let's look at an example using AWS Secrets Manager. The first step is to store your database credentials in AWS Secrets Manager, which you can do using the AWS Web Console, as shown in Figure 6-2.


Secrets Manager
eu-west-1.console.aws.amazon.com/secretsmanager/home?region=eu-west-1#/newSecret?step=selectSecret
aws
Services
Resource Groups
Ireland
Support
Secret type
Step 2
Name and description
Step 3
Configure rotation
Step 4
Review
Feedback
English (US)
© 2008 - 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Store a new secret

Select secret type Info

( ) Credentials for RDS database
( ) Credentials for DocumentDB database
( ) Credentials for Redshift cluster
( ) Credentials for other database
[✓ Other type of secrets (e.g. API key)]

Specify the key/value pairs to be stored in this secret Info

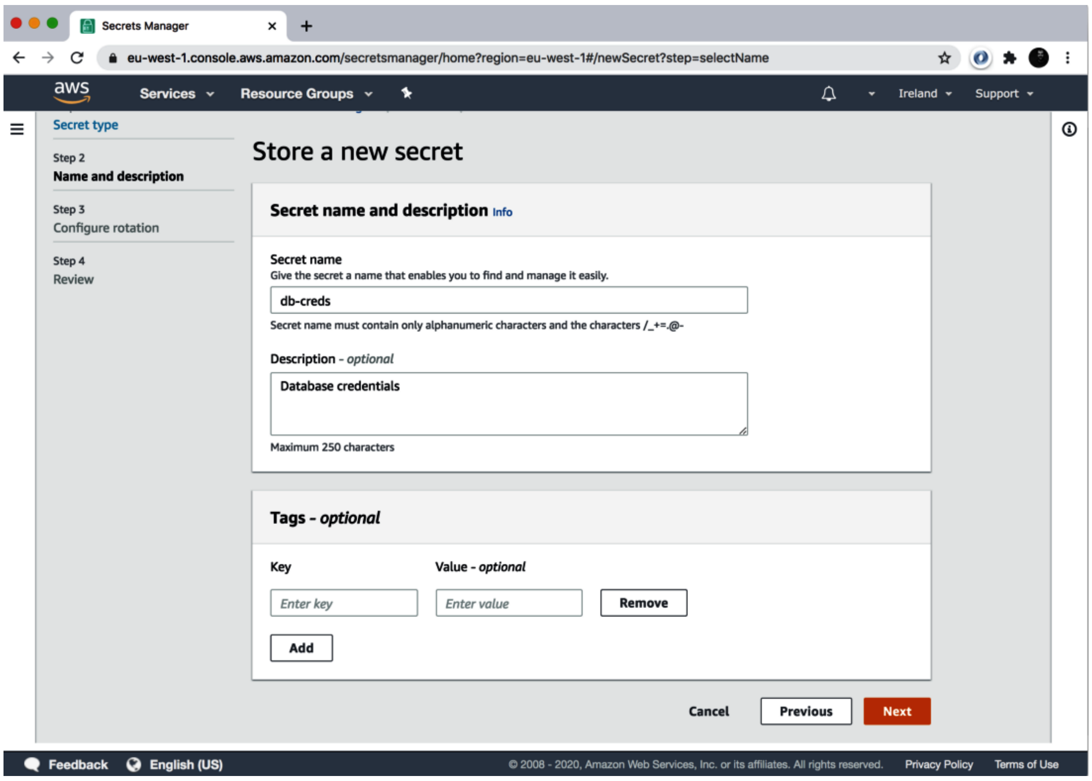
Secret key/value | Plaintext

```
{
 "username": "admin",
 "password": "password"
}
```

Figure 6-2. Store secrets in JSON format in AWS Secrets Manager.

Note that the secrets in Figure 6-2 are in a JSON format, which is the recommended format for storing data in AWS Secrets Manager.

Go to the next step, and make sure to give the secret a unique name, such as `db-creds`, as shown in Figure 6-3.


Secrets Manager
eu-west-1.console.aws.amazon.com/secretsmanager/home?region=eu-west-1#/newSecret?step=selectName
aws
Services
Resource Groups
Ireland
Support
Secret type
Step 2
Name and description
Step 3
Configure rotation
Step 4
Review
Feedback
English (US)
© 2008 - 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Store a new secret

Secret name and description Info

Secret name
Give the secret a name that enables you to find and manage it easily.
db-creds
Secret name must contain only alphanumeric characters and the characters /_+=.@-

Description - *optional*
Database credentials
Maximum 250 characters

Tags - *optional*

Key: Enter key
Value - *optional*: Enter value
Remove
Add

Cancel Previous Next

Figure 6-3. Give the secret a unique name in AWS Secrets Manager.

Click Next and Store to save the secret. Now, in your Terraform code, you can use the `aws_secretsmanager_secret_version` data source to read the `db-creds` secret:

```
data "aws_secretsmanager_secret_version" "creds" {
  secret_id = "db-creds"
}
```

Since the secret is stored as JSON, you can use the `jsondecode` function to parse the JSON into the local variable `db_creds`:

```
locals {
  db_creds = jsondecode(
    data.aws_secretsmanager_secret_version.creds.secret_string
  )
}
```

And now you can read the database credentials from `db_creds` and pass them into the `aws_db_instance` resource:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # Pass the secrets to the resource
  username = local.db_creds.username
  password = local.db_creds.password
}
```

Using secret stores has the following advantages:

- Keep plain-text secrets out of your code and version control system.
- Everything is defined in the code itself. There are no extra manual steps or wrapper scripts required.
- Storing secrets is easy, as you typically can use a web UI.
- Secret stores typically support rotating and revoking secrets, which is useful in case a secret gets compromised. You can even enable rotation on a scheduled basis (e.g., every 30 days) as a preventative measure.
- Secret stores typically support detailed audit logs that show you exactly who accessed what data.
- Secret stores make it easier to standardize all your secret practices, as they enforce specific types of encryption, storage, access patterns, etc.

Using secret stores has the following drawbacks:

- Since the secrets are not versioned, packaged, and tested with your code, configuration errors are more likely, such as adding a new secret in one environment (e.g., staging) but forgetting to add it in another environment (e.g., production).
- Most managed secret stores cost money. For example, AWS Secrets Manager charges $0.40 per month for each secret you store, plus $0.05 for every 10,000 API calls you make to store or retrieve data. A typical usage pattern, where you have several dozen secrets stored across several environments and a handful of apps that read those secrets during boot, usually costs around $10–$25/month. With larg-

er deployments, where you have dozens of apps reading hundreds of secrets, the price can go up to hundreds of dollars per month.

- If you're using a self-managed secret store such as HashiCorp Vault, then you're both spending money to run the store (e.g., paying AWS for 3–5 EC2 Instances to run Vault in a highly available mode) and spending time and money to have your developers deploy, configure, manage, update, and monitor the store. Developer time is very expensive, so depending on how much time they have to spend on setting up and managing the secret store, this could cost you thousands of dollars per month.
- Retrieving secrets is harder, especially in automated environments (e.g., an app booting up and trying to read a database password), as you have to solve how to do secure authentication between multiple machines.
- Integrating with automated tests is harder, as much of the code you're testing now depends on a running, external system that either needs to be mocked out or have test data stored in it.

State Files and Plan Files

There are two more places where you'll come across secrets when using Terraform:

- State files
- Plan files

State files

Hopefully, this chapter has convinced you to not store your secrets in plain text and provided you with some better alternatives. However, something that catches many Terraform users off guard is that, no matter which technique you use, *any secrets you pass into your Terraform resources and data sources will end up in plain text in your Terraform state file!*

For example, no matter where you read the database credentials from—environment variables, encrypted files, a centralized secret store—if you pass those credentials to a resource such as `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
```

```
    engine              = "mysql"
    allocated_storage   = 10
    instance_class      = "db.t2.micro"
    skip_final_snapshot = true
    db_name             = var.db_name

    # Pass the secrets to the resource
    username = local.db_creds.username
    password = local.db_creds.password
  }
```

then Terraform will store those credentials in your *terraform.tfstate* file, in plain text. This has been an open issue since 2014, with no clear plans for a first-class solution. There are some workarounds out there that can scrub secrets from your state files, but these are brittle and likely to break with each new Terraform release, so I don't recommend them.

For the time being, no matter which of the techniques discussed you end up using to manage secrets, you must do the following:

Store Terraform state in a backend that supports encryption

> Instead of storing your state in a local *terraform.tfstate* file and checking it into version control, you should use one of the backends Terraform supports that natively supports encryption, such as S3, GCS, and Azure Blob Storage. These backends will encrypt your state files, both in transit (e.g., via TLS) and on disk (e.g., via AES-256).

Strictly control who can access your Terraform backend

> Since Terraform state files may contain secrets, you'll want to control who has access to your backend with *at least* as much care as you control access to the secrets themselves. For example, if you're using S3 as a backend, you'll want to configure an IAM Policy that solely grants access to the S3 bucket for production to a small handful of trusted devs, or perhaps solely just the CI server you use to deploy to prod.

Plan files

You've seen the `terraform plan` command many times. One feature you may not have seen yet is that you can store the output of the plan command (the "diff") in a file:

```
$ terraform plan -out=example.plan
```

The preceding command stores the plan in a file called *example.plan*. You can then run the `apply` command on this saved plan file to ensure that Terraform applies *exactly* the changes you saw originally:

```
$ terraform apply example.plan
```

This is a handy feature of Terraform, but an important caveat applies: just as with Terraform state, *any secrets you pass into your Terraform resources and data sources will end up in plain text in your Terraform plan files!* For example, if you ran `plan` on the `aws_db_instance` code, and saved a plan file, the plan file would contain the database username and password, in plain text.

Therefore, if you're going to use plan files, you must do the following:

Encrypt your Terraform plan files

> If you're going to save your plan files, you'll need to find a way to encrypt those files, both in transit (e.g., via TLS) and on disk (e.g., via AES-256). For example, you could store plan files in an S3 bucket, which supports both types of encryption.

Strictly control who can access your plan files

> Since Terraform plan files may contain secrets, you'll want to control who has access to them with *at least* as much care as you control access to the secrets themselves. For example, if you're using S3 to store your plan files, you'll want to configure an IAM Policy that solely grants access to the S3 bucket for production to a small handful of trusted devs, or perhaps solely just the CI server you use to deploy to prod.

Conclusion

Here are your key takeaways from this chapter. First, if you remember nothing else from this chapter, please remember this: you should *not* store secrets in plain text.

Second, to pass secrets to providers, human users can use personal secrets managers and set environment variables, and machine users can use stored credentials, IAM roles, or OIDC. See Table 6-2 for the trade-offs between the machine user options.

Table 6-2. A comparison of methods for machine users (e.g., a CI server) to pass secrets to Terraform providers

	Stored credentials	**IAM roles**	**OIDC**
Example	CircleCI	Jenkins on an EC2 Instance	GitHub Actions
Avoid manually managing credentials	x	✓	✓
Avoid using permanent credentials	x	✓	✓
Works inside of cloud provider	x	✓	x
Works outside of cloud provider	✓	x	✓
Widely supported as of 2022	✓	✓	x

Third, to pass secrets to resources and data sources, use environment variables, encrypted files, or centralized secret stores. See Table 6-3 for the trade-offs between these different options.

Table 6-3. A comparison of methods for passing secrets to Terraform resources and data sources

	Environment variables	**Encrypted files**	**Centralized secret stores**
Keeps plain-text secrets out of code	✓	✓	✓
All secrets management defined as code	x	✓	✓
Audit log for access to encryption keys	x	✓	✓
Audit log for access to individual secrets	x	x	✓
Rotating or revoking secrets is easy	x	x	✓
Standardizing secrets management is easy	x	x	✓
Secrets are versioned with the code	x	✓	x
Storing secrets is easy	✓	x	✓
Retrieving secrets is easy	✓	✓	x
Integrating with automated test-	✓	x	x

	Environment variables	Encrypted files	Centralized secret stores
ing is easy			
Cost	0	$	$$$

And finally, fourth, no matter how you pass secrets to resources and data stores, remember that Terraform will store those secrets in your state files and plan files, in plain text, so make sure to always encrypt those files (in transit and at rest) and to strictly control access to them.

Now that you understand how to manage secrets when working with Terraform, including how to securely pass secrets to Terraform providers, let's move on to Chapter 7, where you'll learn how to use Terraform in cases where you have multiple providers (e.g., multiple regions, multiple accounts, multiple clouds).

1 Note that in most Linux/Unix/macOS shells, every command you type is stored on disk in some sort of history file (e.g., *~/.bash_history*). That's why the `export` commands shown here have a leading space: if you start your command with a space, most shells will skip writing that command to the history file. Note that you might need to set the `HISTCONTROL` environment variable to "ignoreboth" to enable this if your shell doesn't enable it by default.

2 By default, the instance metadata endpoint is open to all OS users running on your EC2 Instances. I recommend locking this endpoint down so that only specific OS users can access it: e.g., if you're running an app on the EC2 Instance as user *app*, you could use `iptables` or `nftables` to only allow *app* to access the instance metadata endpoint. That way, if an attacker finds some vulnerability and is able to execute code on your instance, they will only be able to access the IAM role permissions if they are able to authenticate as user *app* (rather than as any user). Better still, if you only need the IAM role permissions during boot (e.g., to read a database password), you could disable the instance metadata endpoint entirely after boot, so an attacker who gets access later can't use the endpoint at all.

3 At the time this book was written, OIDC support between GitHub Actions and AWS was fairly new and the details subject to change. Make sure to check the latest GitHub OIDC documentation for the latest updates.