- 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