```
$ 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 testing is easy	✓	x	x
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.