after a build)! If you can still access this file after a failed deployment, as soon as internet connectivity is restored, you can push this file to your remote backend (e.g., to S3) using the `state push` command so that the state information isn't lost:

```
$ terraform state push errored.tfstate
```

Errors releasing locks

Occasionally, Terraform will fail to release a lock. For example, if your CI server crashes in the middle of a `terraform apply`, the state will remain permanently locked. Anyone else who tries to run `apply` on the same module will get an error message saying the state is locked and showing the ID of the lock. If you're absolutely sure this is an accidentally leftover lock, you can forcibly release it using the `force-unlock` command, passing it the ID of the lock from that error message:

```
$ terraform force-unlock <LOCK_ID>
```

Deployment server

Just as with your application code, all of your infrastructure code changes should be applied from a CI server and not from a developer's computer. You can run `terraform` from Jenkins, CircleCI, GitHub Actions, Terraform Cloud, Terraform Enterprise, Atlantis, or any other reasonably secure automated platform. This gives you the same benefits as with application code: it forces you to fully automate your deployment process, it ensures deployment always happens from a consistent environment, and it gives you better control over who has permissions to access production environments.

That said, permissions to deploy infrastructure code are quite a bit trickier than for application code. With application code, you can usually give your

CI server a minimal, fixed set of permissions to deploy your apps; for example, to deploy to an ASG, the CI server typically needs only a few specific `ec2` and `autoscaling` permissions. However, to be able to deploy arbitrary infrastructure code changes (e.g., your Terraform code might try to deploy a database or a VPC or an entirely new AWS account), the CI server needs arbitrary permissions—that is, admin permissions. And that's a problem.

The reason it's a problem is that CI servers are (a) notoriously hard to secure,[5] (b) accessible to all the developers at your company, and (c) used to execute arbitrary code. Adding permanent admin permissions to this mix is just asking for trouble! You'd effectively be giving every single person on your team admin permissions and turning your CI server into a very high-value target for attackers.

There are a few things you can do to minimize this risk:

Lock the CI server down

> Make it accessible solely over HTTPs, require all users to be authenticated, and follow server-hardening practices (e.g., lock down the firewall, install fail2ban, enable audit logging, etc.).

Don't expose your CI server on the public internet

> That is, run the CI server in private subnets, without any public IP, so that it's accessible only over a VPN connection. That way, only users with valid network access (e.g., via a VPN certificate) can access your CI server at all. Note that this does have a drawback: webhooks from external systems won't work. For example, GitHub won't automatically be able to trigger builds in your CI server; instead, you'll need to configure your CI server to poll your version control system for updates. This is a small price to pay for a significantly more secure CI server.

Enforce an approval workflow