IT 351 Cloud Security and DevSecOps

Lab 3

Fall 2021 – Professor Josh Edgar

Now that you've learned the basics of GitLab and have executed a CI/CD job using a GitLab Runner, we will use our third Lab to work with a tool by HashiCorp, called "Terraform." Terraform is hypervisor- and cloud-agnostic (meaning it can launch resources in on-prem VMware systems, or in any cloud – AWS, Azure, GCP, etc.)

Before you begin:

- You should have already completed Lab 2 successfully.
- Watching the Panopto walkthroughs for this Lab and all of the ones that follow is highly recommended from this point, as the professor adds valuable information to the demonstrations that might not be conveyed in these written guides.

What you'll achieve:

- Manually deploy the dependencies needed for Terraform to operate
- Write a Terraform configuration that will deploy an EC2 instance using automation
- Write GitLab CI/CD instructions that will lay out the stages needed for a Terraform deployment

What you'll turn in with this lab:

- A screenshot of the instance that your pipeline will create (you will destroy it at the end of the Lab to keep costs low), uploaded to the Lab 3 assignment page on Canvas.
    - Don't try to cut corners by just deploying the instance manually – The professor will still be checking your GitLab CI/CD logs to verify that Terraform created the instance!

Task 1: Deploy an S3 Bucket for Terraform State

The way that Terraform keeps track of which cloud resources it's deployed – and thus limits which ones its allowed to destroy/rebuild/etc. – is through the use of "state files," a small JSON file that lists out all of the resources it manages.

If we were just running Terraform directly from an EC2 without implementing CI/CD, this file would just be stored on the local storage of that instance. However, since we will be using CI/CD through our GitLab Runners and splitting up the various Terraform activities into separate stages and jobs – and since each job spins up a temporary Docker container that is destroyed by the end of the job – we need to store our state file somewhere where it can be repeatedly accessed outside of Docker. For these reasons, we will store our state file in an S3 Bucket.

1. Log in to the AWS Management Console. Navigate to the S3 Console.

2. Create an S3 Bucket in the U.S. East (N. Virginia) Region. (Remember that S3 is strange and the Region selector at the top right will always show "Global," even though individual Buckets must still be deployed to a specific Region.)
    a. Name the Bucket something like "your-name-tfstate" (replacing "your-name," of course).
    b. You can leave all privacy settings as their defaults (block all public access). We don't want to advertise our Terraform configurations on the open Internet!

Make sure you don't forget your Bucket name and Region code ("us-east-1" if you deployed in Northern Virginia). We'll need them in a future Task.

Task 2: Create IAM Access Keys and Add GitLab Variables

Terraform lacks the ability to log into our AWS accounts with our e-mail addresses and passwords. Instead, it relies on the AWS Application Programming Interface (API) to carry out cloud deployment tasks, and access to the AWS API is governed by access and secret keys created by Identity and Access Management (IAM).

In the real-world, you'd want to first create a separate IAM User with limited privileges so that the API access keys you generate don't have root access to your entire AWS account. However, for the purposes of this Lab, we assume you don't have any mission-critical services running in your personal AWS account, and we'll skip this step for brevity. Just remember that, in a real enterprise cloud environment, it's best practice to create a limited IAM User first, and then generate access keys for it – this way, you're not giving away the keys to the whole city, should they ever fall into the wrong hands.

3. In the AWS Management Console, click the dropdown at the top right that contains your name. Click "My Security Credentials."
4. Expand the section entitled "Access keys (access key ID and secret access key). Click "Create New Access Key." This will instantly create a new pair of access and secret keys for your API access.
5. Expand the "Show Access Key" link in the pop-up box and copy both keys to a text file where you can access them again later. Save this somewhere safe on your computer – *not* in Git where other people could see them.

You do not need to download the key file – we just need the key IDs themselves. **Do not lose these!** Similar to SSH key pairs when you launch an EC2 instance, if you ever lose the secret key, you cannot recover it, and you'll have to start over.

6. Log in to GitLab in your web browser. Navigate to your project, and in the left sidebar, navigate to Settings > CI/CD.

7. Expand the “Variables” section. We need to add our access and secret keys as separate variables, which GitLab will pass to our Runner environment every time we launch a pipeline. Click “Add variable.”
    a. For the first variable, the key should be “AWS_ACCESS_KEY_ID” (without quotes), and the value should be the Access Key ID you captured in step 5. Uncheck the “protect variable” and/or “mask variables” boxes if they are checked, and then click “Add variable” to save the variable.
    b. For the second variable, follow the same steps as above, except the key should be “AWS_SECRET_ACCESS_KEY” and the value should be your Secret Access Key.

Again, in an enterprise environment, we would probably protect and mask these variables so that our keys never end up in the wrong hands – but in this scenario, only you and the professor have access to your GitLab project, and these options require additional configuration of our repos to use – not the best use of our time for a simple Lab.

Task 3: Write Terraform Configuration Files

Terraform will read any “.tf” format files (written in HashiCorp Configuration Language, or “HCL”) that are in your repo when GitLab provides those files to it during a pipeline’s execution. Its interpreter is generally smart enough to look at any “.tf” files of any name and figure out in which order cloud resources should be deployed.

There are two files that Terraform must have, though: “provider.tf,” which contains crucial information about which cloud service provider and Region to deploy within; and “main.tf,” which should contain the details of at least one resource to deploy, and then it can look around the rest of the repo if more information is needed. For this Task, we will simply create those two (2) required files and write very simple HCL to create an EC2 instance with our CI/CD pipeline.

8. Download the Terraform Sample Code that was linked from the Lab assignment page on Canvas.
9. Open VS Code. Ensure that you are working in your Git repo under the “Explorer” tab.
10. Create a new file called “provider.tf” (without quotes). From the sample code, copy the portion that is intended for “provider.tf” into this file. The professor will explain what this code is doing in the Panopto walkthrough. Save the file.
11. Repeat step 10, but for the “main.tf” content that was also provided in the sample code. Adjust the Name tag for the instance code to reflect your name, e.g. “ProfEdgarFirstInstance,” replacing “ProfEdgar” with your own name. Save the file.

Don’t commit or push these files just yet, but do take some time to review the Terraform code and get a basic understanding for what you’re asking Terraform to do. It is hopefully clear that Terraform will be creating an EC2 instance with the parameters defined by our short block of code.

Task 4: Update Our GitLab CI/CD File to Use Terraform

Although we've written Terraform code into our repo, our GitLab Runner won't know what to do with it until we give it specific instructions about jobs we want our pipeline to contain. In this Task, we will make significant changes to our old "hello world" CI/CD file, laying out the four (4) stages we will need for a successful Terraform pipeline: validate, plan, apply and destroy. Each stage will call upon the "hashicorp/terraform" Docker image, publicly available in the Docker Hub, meaning each job will use a temporary container that is pre-loaded with the Terraform dependencies.

The "validate" stage will check the syntax of our Terraform code and make sure there are no obvious errors locally. It won't prevent any potential conflicts from occurring in AWS, because it hasn't connected to the API yet. If the syntax looks good, we will advance to the "plan" stage, which connects to AWS, evaluates our current environment, and lays out a plan for how it will take our code and turn it into infrastructure – infrastructure as code (IaC)!

Any potential conflicts in our cloud environment will be called out by the plan. If there are issues that Terraform can't self-resolve by removing/replacing other Terraform-managed resources, the plan will "fail" and you will receive feedback as to what needs to be fixed. Otherwise, the plan should announce its intentions to create your infrastructure, at which time you can move to the "apply" stage to execute that plan. Finally, when you're done with these resources, you can use the "destroy" stage as a "reverse apply" of sorts, terminating the infrastructure that Terraform created. We will use a `when` statement in our CI/CD file to make our "apply" and "destroy" jobs triggered "manually," so that we don't accidentally make big changes to our cloud environment without first reviewing exactly what is going to happen.

Each of these stages/jobs requires Terraform to be initialized before the service can be used, which we will write as a function and then call that function from our `before_script` section of each job. You'll also notice that some `artifacts` are mentioned in the file – this is how GitLab saves files temporarily, making them available to other jobs in the pipeline even after our Docker containers are shut down. This is what makes our plan file from the "plan" job available to the "apply" job. **It is highly recommended to watch the Panopto walkthrough so all of this can be better explained verbally!**

12. Download the GitLab Sample Code that was linked from the Lab assignment page on Canvas.
13. In your Git repo in VS Code, open your ".gitlab-ci.yml" file.
14. Remove the existing content from your "hello world" job and replace it with the content from the sample code file.
15. Pay special attention to the `.terraform-init` function near the top of the code. Make sure to change the value of "bucket=" to reflect the name of your S3 Bucket created in step 2. Save the file.

16. Commit your changes from Tasks 3 and 4 to your Git repo with an appropriate comment. Push your changes to GitLab.
17. Navigate back to GitLab in your browser. On the left sidebar, navigate to CI/CD > Pipelines. You should see a pipeline for your recent commit – select it. Notice how you now have multiple jobs split across four (4) stages. You can click on the "tf-val" job to watch the validation stage in action. Once it succeeds, it will automatically start working on the "tf-plan" job, which you can select by changing the dropdown on the right side of the terminal output to the "plan" stage, and then clicking on the name of the only job under that stage. If the plan succeeds, you can then use the dropdown again to navigate to the "apply" stage and manually trigger the "tf-apply" job. (This can also be done by pressing the "play" icon on the job from the pipeline overview page.)

If the apply job completes successfully, you should see your new instance appear in the AWS Management Console! Take a screenshot of that, and turn it in on Canvas. You've just built an IaC pipeline using Terraform!

Task 5: Destroy What You've Just Created

Let's see just how easy it is to undo what we've created by using Terraform's destroy job.

Two important notes about destroying things with Terraform: One, you won't get an opportunity to approve its plan. If you start the destroy job, the horse has already left the barn and your infrastructure will be destroyed, whether you meant for it to happen or not. This is why most enterprises find creative ways to conceal the destroy job unless certain variables are set. Two, the destruction is not limited to what you've created with the most recent pipeline execution. Terraform will destroy everything that your pipeline ever created, past or present, based on what was contained within the state file.

18. Use the dropdown again to navigate to the "destroy" stage and "tf-destroy" job.
19. Trigger the job and watch as your newly-created instance is destroyed!

Task 6 (Optional): Stop Your Runner Instance

In the interest of saving students some money, we can actually "stop" (**not** terminate!) our GitLab Runner instance when we aren't using it. Your EBS Volume, which contains the operating system and binaries that the Runner needs to function, will continue to persist and you will continue being charged for it, but you will not be charged for the CPU usage of your EC2 instance running when it's unused. This amounts to about a 50% savings.

It's entirely up to you if you want to do it, but as long as you ran the systemctl enable commands from Lab 2, which added the GitLab Runner and Docker services to the startup processes list, then whenever you start your Runner instance back up, those services should

automatically restart, GitLab should see them, and you can continue working on future Labs or Projects.

20. In the AWS Management Console, return to the EC2 Instances table. Highlight your GitLab Runner instance, go to “Instance state,” and choose “Stop instance.” When it’s time to do a future Lab, you can use this same menu to start it back up again.