

IAC-CLI Blogpost & project description

By Guido Borst, Student at Radboud University

Abstract

This blogpost presents the Infrastructure as Code Command Line Interface (IaC CLI), an infrastructure deployment orchestration for small lab environments without a CI/CD pattern dependency. It uses Packer, Terraform and Ansible for deploying and configuring Virtual Machines (VMs), while using Vault for managing sensitive variables. It consists of two scripts, one for creating templates, and one for deploying the templates and configuring the deployed VMs. Its primary aim is to allow Blue and Red Teams to quickly spin up new lab environments.

Introduction

Deploying virtual labs consisting of multiple different Virtual Machines is not something most people have to do often. However, deploying virtual labs can be very beneficial for cyber security experts. These labs can be used in Offensive Security for example, where Red Teams can use these labs to simulate the target infrastructure, and test the effectiveness of certain attack vectors before applying them on the target network.

In light of this, I have done my internship at [redacted], where I have researched the aspect of deploying virtual labs and created the Infrastructure as Code Command Line Interface (IaC CLI), that aims to simplify this matter, while still being flexible and extendable (<https://github.com/Guido-Borst/IaC-CLI/>). The IaC CLI is an infrastructure deployment orchestration for small lab environments without a CI/CD pattern dependency. It is a user-friendly way of deploying new virtual machines in VMware vSphere using Terraform, and allows for applying custom configurations with the use of Ansible. The images that are later deployed to VMs are created using Packer, and in the case of Windows, are fully sysprepped.

Background

Of course, we're not the first to think about how to deploy these virtual labs. One way to deploy such a lab is by using a CI/CD pipeline. When a new lab is needed, the pipeline is started and it spits out a new lab. This is not very flexible however, because changing the configuration of the pipeline is usually needed to produce different labs. Another way to do it is manually, where the images are captured from already installed VMs, and these images are then later deployed to new networks. This involves a lot of manual work, and changing the configuration of the image or the VM is very labor intensive. This led us to develop a

project that aims to be both flexible and customizable, with a minimum amount of manual input required, and which takes security into account as well.

Goals

This project has been built with the following goals in mind:

- **Built for small environments:** This project is aimed to be used in small environments. While it could work for larger environments as well, those will typically make use of CI/CD pipelines and would not benefit as much from this solution. It should be fairly easy to get up and running
- **No CI/CD dependency:** In relation with the goal mentioned above, our project should not depend on a CI/CD pipeline, it should be fully standalone.
- **Modular:** Another goal we set ourselves is that our project should be modular: the individual components should not have a hard dependency on each other, it should be possible to swap out these modules for others as long as they provide similar functionality.
- **Easy to extend:** Another feature that this project should have, is that it should be easy to extend upon the functionality of the project. In particular, it should be possible to add custom scripts which will be presented to the user with the option to select which scripts will get executed during the process.
- **Security of credentials:** The last goal is that there should be a good amount of security when it comes to handling sensitive variables such as credentials. This project should handle these sensitive variables in a better way than storing them plain-text in configuration files.

Design

The *laC CLI* is an *infrastructure deployment orchestration CLI without a CI/CD pattern dependency for small lab environments*. It is a modular and easily extendable way to securely deploy infrastructure, with very few user input required and no CI/CD dependency. The *laC CLI* is built mainly to be used with vSphere, although modifying it to work with AWS should be easy to do.

From a high-level overview the project has three stages: creating VM-templates to be used for deploying, deploying the templates to VMs, and configuring the deployed VMs. The different stages and the tools used are:

1. **Building templates:** The first stage is to create VM-templates from installation files to be used for deploying in the second phase. This is described in Section [Packer](#). The main tool used during this stage is:
 - **Packer**
2. **Deploying templates:** The second stage is to deploy the templates and create the necessary supporting infrastructure, like port groups/virtual networks. This is described in Section [Terraform](#). The main tool used during this stage is:
 - **Terraform**

3. **Configuring VMs:** The final stage is to configure the deployed VMs. This is described in Section [Ansible](#). The main tool used during this stage is:
 - **Ansible**

Finally, there is a fourth tool that we use throughout the project, namely **Vault**. Vault is used to securely manage sensitive variables like passwords used in the script and is described in Section [Vault](#). It is not confined to a particular stage, but can be used in all of the three stages. Currently it is only used in stage 2 and 3, because we didn't have time to integrate it with Packer. That is a topic for future work.

All these components are integrated into two scripts: one that uses Packer to create the templates and another to deploy and configure infrastructure using Terraform, Ansible and Vault. The rationale behind using two scripts instead of one or three is that the templates should only be created and updated once in a while (say monthly or quarterly), but deploying infrastructure could happen much more often, depending on the use case. And since stage 2 and 3 are strongly connected, we combined the deployment and configuration of VMs into one script, as this is much more intuitive and user friendly.

We will describe the two scripts along with the used tools in the sections below.

Script one: Creating templates

The groundwork for this part is laid by a GitHub repo by VMware: [GitHub - vmware-samples/packer-examples-for-vsphere](#). This repo includes configuration files for Packer to build templates for a lot of different distributions, exactly what we need. It includes Windows 10 and 11, Windows Server 2019 and 2022, and various versions of Ubuntu, Debian, Red Hat Enterprise and more.

Since the first part of our project is based upon this repo, we follow their instructions to get started: [Getting Started - Requirements](#).

The directory structure for this part of the project looks as follows:

```
builds
|
|-- linux
|   |-- almalinux
|   |   |-- 8
|   |   |   |-- data
|   |   |   |   |-- ks.pkrtpl.hcl
|   |   |   |   |-- linux-almalinux.auto.pkrvars.hcl
|   |   |   |   |-- linux-almalinux.pkr.hcl
|   |   |   |   |-- variables.pkr.hcl
|   |-- ... (other Linux distributions)
|
|-- windows
|   |-- desktop
|   |   |-- 10
|   |   |   |-- data
```

```

|   |   |   |   |-- autounattend.pkrtpl.hcl
|   |   |   |   |-- variables.pkr.hcl
|   |   |   |   |-- windows.auto.pkrvars.hcl
|   |   |   |   |-- windows.pkr.hcl
|   |-- server
|   |   |   |   ... (other Windows versions)
|
config
|-- Ansible.pkrvars.hcl.example
|-- vsphere.pkrvars.hcl.example
|-- ... (other config files)
|
build.sh

```

Packer

The script `build.sh` is a bash script that lists the possible templates that can be built. After selecting an option, say Windows 10, Packer will initialize the directory corresponding to the chosen option, and build the template. It uses the `variables.pkr.hcl` for variable definition, alongside with `{distro}.auto.pkrvars.hcl` for initialisation of these variables. The `{distro}.pkh.hcl` lists the steps taken for creating the template. The file for automated setup is in the `data` sub-directory, and packer will use that to set up the OS without user interaction. For more in-depth information about how these configuration files work, see the official Packer documentation: [HCL Templates | Packer | HashiCorp Developer](#)

Packer starts with creating a new VM on vSphere with the proper resources, and connects the iso image of the selected OS to the VM. Then, it'll automatically go through the setup of the system using the `autounattend` file for Windows or the `user-data` or `ks` files for Linux.

For Windows, it installs [Chocolatey](#) as well. When creating Windows templates, the last step of the setup is generalizing Windows using the built-in sysprep tool. This ensures that when deploying the template to multiple VMs, each VM will have a different GUID and SID. This is necessary when Windows PCs join an Active Directory, as PCs with the same SID and GUID will conflict with each other. This is done by uploading an autounattend file executing sysprep after setting some variables in the register.

Doing it with Packer during template creation instead of using a Terraform-module ensures that deploying the VMs takes less time, as part of the sysprep process is already done during the creation of the template. Another benefit of this method is that it doesn't rely on specific modules from Terraform, which helps the modularity-goal. Also, when templates are deployed manually instead of using Terraform, VMs will still have different identifiers now.

Because Windows 11 needs a TPM device and the vSphere server we had access to while developing this project didn't have a vTPM card at the time, we'll set a registry value that bypasses this check which allows the setup to continue. Ideally we would use the vTPM card, so this is something that can be improved upon in the future.

Script two: Deploy and configure

The second script is built in Python 3.10, although it should work for later versions as well. It asks the user a few general questions first, and then asks the user which templates (and how many of them) should be deployed. It also asks the user to which port group (virtual network) the VM should be connected, which Ansible playbooks should be run and which type of OS it is (although it'll guess the latest based on the name of the template). After all these questions are answered, it will deploy the selected VMs using Terraform and execute the selected Ansible playbooks.

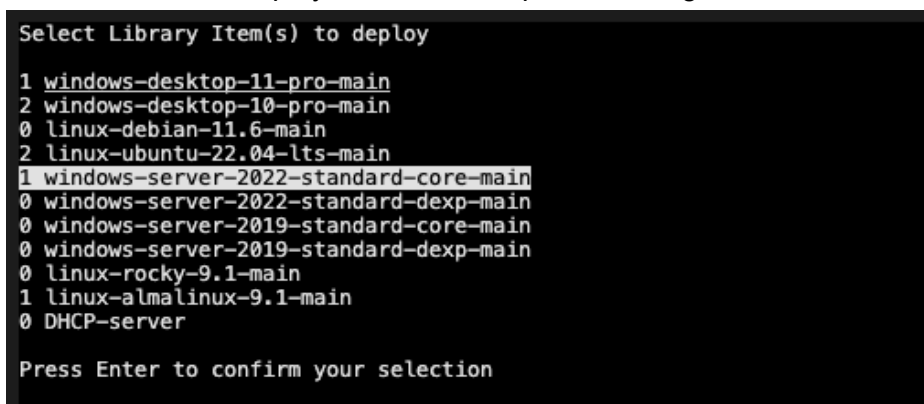
The script questions

The first thing the script will ask is the name for the current project. This is used for naming the VMs, so that different deployments are easily distinguished in vSphere. The next question is in which directory the temporary working directories for Terraform should be created. This is important because in here temporary files containing plain-text passwords are written to, so ideally we would use a ramdisk directory for this.

After that, the script asks whether the user wants to create a new port-group/virtual network in vSphere. If the user chooses to do this, the script will ask for the name and the VLAN ID of the port group, along with whether enabling promiscuous mode should be allowed on this port group or not. Finally, the user is asked whether they want a DHCP server to be set up in this port group.

Because Terraform needs the name of the library item (the template that we created with the first script) that it wants to deploy, we first list all available library items and lets the user select which, and how many, of those templates it wants to deploy. To do this, we used a project on Github ([vsphere-automation-sdk-python](https://github.com/vmware/vsphere-automation-sdk-python)) from VMware itself to connect to the vSphere server and get a list of library items. Here we see the first usage of HashiCorp Vault: we use the HashiCorp Vault API to request the login details for the vSphere server. For more information about Vault, see section [Vault](#).

To let users choose, we use the `curses` library to create a menu where they can specify the amount of VMs to deploy from each template, see Figure 1 below.



```
Select Library Item(s) to deploy
1 windows-desktop-11-pro-main
2 windows-desktop-10-pro-main
0 linux-debian-11.6-main
2 linux-ubuntu-22.04-lts-main
1 windows-server-2022-standard-core-main
0 windows-server-2022-standard-dexp-main
0 windows-server-2019-standard-core-main
0 windows-server-2019-standard-dexp-main
0 linux-rocky-9.1-main
1 linux-almalinux-9.1-main
0 DHCP-server

Press Enter to confirm your selection
```

Figure 1: Screenshot of script two asking the user to select which and how many library items should be deployed.

After selecting the VMs, the user is asked for each VM to which port-group the VM should be connected, which Ansible playbooks should be run and whether the VM OS is Windows or Linux (although the script will try to detect this based on the name).

All these settings are saved, only then Terraform will proceed to deploy the VMs and, if applicable, create the port group.

Terraform

Now that we have created the templates using Packer, and have answered all the questions of the second script, it is now time to deploy the selected VMs using Terraform.

Terraform works similar to Packer in that it requires similar configuration files as Packer. We use the file `variables.tf` for variable declaration, along with `terraform.tvvars` to instantiate these variables. It then uses all other `.tf` files in the directory that Terraform is initiated in to deploy whatever is specified in the `.tf` files. This means that if we copy `variables.tf`, `terraform.tvvars` and `deploy-ovf-linux.tf` to a new directory and run `'terraform apply'`, Terraform will try to deploy the Linux template specified by the variables in `terraform.tvvars`. This works similarly for the files `deploy-ovf-windows.tf` and `create-host-port-group.tf`.

Given that Terraform is an external program we need to execute commands on the machine that the script is running on. Fortunately, this is easy to do with the `subprocess` library. Given a list of commands and a directory, it will run these commands from that directory, printing the output to the console.

With this in mind, deploying a VM is as easy as copying the right Terraform files to a new temporary directory that we create inside the configured working directory (see section [The script questions](#) for more info) and setting the variables inside `terraform.tvvars` correctly for that specific VM. Then we initialize Terraform in this new directory, execute `terraform apply`, wait for it to finish and at last remove the directory. This is repeated for each VM.

When creating a new port group with Terraform, it is important that the port group has a unique name. If there already exists a port group on this vSphere host with the same name, creating a new one will fail. The script will continue though, as this should not lead to invalid configurations for VMs, as they will just get connected to the already existing port group.

If the user has selected the option to deploy a DHCP server in the newly created port group, a Linux template with the name `DHCP-server` will be deployed. The deployment function will be informed of this, enabling adjustments to deployment parameters like RAM and CPU allocation. This is because a simple DHCP server doesn't need that much resources, and it would be a waste to allocate more resources than needed.

Also, since Terraform supports gracefully aborting and cleaning up on receiving a SIGINT (CTRL+C) signal, we catch these signals and forward them to the subprocess that is

currently running, temporarily overwriting the default signal handler of Python that just exits the whole script.

Ansible

After the VMs are deployed, we have the option to run configuration scripts on them to, for example, make sure that all VMs are part of the same Active Directory.

To run these configuration scripts, called playbooks, we first need an inventory-file or -folder to tell it which hosts to run its playbooks on. To make it more versatile, we use a plugin that lets Ansible work together with Terraform. For this plugin to work, the Ansible files should be copied to the same temporary directory that Terraform used to deploy the VM. Since we already use a new temporary directory for each VM, we can simply re-use the directory we already created for Terraform.

Since each VM is assigned a random IP-address after being deployed via Terraform and we want to avoid relying on user input during this stage of the project, we can use the plugin to read Terraform's `.state` file and extract the host information from that.

If the user has selected one or more Ansible playbooks to be run on this VM, the specified playbooks will be copied into the temporary directory, and Ansible will execute the playbooks on the VM.

Vault

By default, Terraform relies on config files such as `terraform.tvvars` for its variables, including sensitive variables like the username and password to connect to vSphere. Because hard-coded passwords in files are neither flexible nor secure, we opted for a different approach. HashiCorp Vault is a tool used for secret management. Because this is developed by the same company as Packer and Terraform (which are also HashiCorp products), there is good support for using them together. Terraform has a plugin that, given a Vault ID and Secret (an API key), connects to this vault and retrieves the stored secrets, making them available for use in the rest of the Terraform configuration file.

While in our project the Vault ID and Secret should be stored in `terraform.tvvars` to make Terraform able to use these, this solution is already better than having all secrets hard-coded into the config file. This is because secrets stored in Vault are more flexible, if they should be changed it is now just a matter of changing them in Vault, and doesn't require updating all config files. Vault also supports revoking API keys and monitoring access to secrets, so if a security breach has been detected but no secrets have been accessed yet, revoking the API key and creating a new one would be enough to ensure safety of the secrets.

Still, users should make sure not to include the `terraform.tvvars` file in any source control, as the Vault ID and Secret that live there would give access to all the secrets inside this vault.

As mentioned in Section [Design](#), Vault is currently only used in the second script, as we did not have enough time for integrating it with packer. The first script, however, would also benefit from using Vault as it currently expects credentials to be inside the config files.

Future work

While this project provides a way for easily building templates, deploying them and configuring the deployed VMs, there is still room for improvements that we haven't been able to implement due to time constraints. These include:

1. **Ramdisk directory:** use the Python script to create a ramdisk in memory, and use that as a temporary directory for writing the Terraform files to. Since Terraform saves a `.state`, which includes the plaintext passwords for vSphere for example, these files are highly sensitive, and preventing them from being written to disk increases security further. Ideally, we would create a ramdisk directory that is only accessible to our script and its subprocesses, as that would prevent other processes from reading the credentials altogether.
2. **Multithreading for Terraform:** currently we loop over all VMs to be deployed sequentially, which means deploying a larger amount of VMs at once will cost quite a bit of time, since each VM will take 2-5 minutes to deploy. Using multiple to run Terraform in parallel would speed up this process significantly.
3. **Packer-Vault integration:** currently Packer doesn't use Vault for accessing secrets, and instead relies on them being specified in its config file. Making sure that Packer uses Vault in a similar way that Terraform does in combination with using a ramdisk directory will ensure that no plaintext credentials are being stored on disk during the execution of these scripts.
4. **Using vTPM:** since Windows 11 normally requires a TPM to work, we are bypassing this requirement by setting a specific register value during template creation. This however decreases the security of the system under test, which might lead to simulations or tests not being valid anymore, as the weakened security of the system does not properly reflect the security one would see in a normal system. Adding full support for Windows 11 can be done by changing the `windows.pkr.hcl` file to use a vTPM module of vSphere, provided the vSphere server is equipped with a vTPM card.
5. **Adding support for AWS:** currently this project has been developed for use with vSphere, but adding support for AWS should be a matter of editing the Packer and Terraform template files to work with AWS instead of vSphere. Examples of this can be easily found online.
6. **Custom scripts in Packer phase:** in regards to the 'Easy to Extend' goal mentioned in [Goals](#), currently the only way to easily add custom scripts is in the Ansible phase, where users can add their own playbooks to the `ansible` directory. Adding support for executing custom scripts during the Packer phase would be great for improving the quality of the template images.
7. **Deployment order:** In certain scenarios, dependencies should be taken into account. For example, deploying a Windows Server that is going to be configured as a Domain Controller and a Windows client that should join the same domain, may result in problems when the Windows client is deployed before the Windows Server. Currently there is no way to specify the order in which Terraform should deploy the

VMs. When implementing multithreading (see option 2), these kinds of dependencies should also be taken into account, requiring the delay of deploying certain VMs until Vms they depend upon are deployed and configured.

8. **Quality of Life improvements:** there are still some QoL improvements that can be made to the scripts, including:
- Show a final overview of all changes to be made to the system before applying them all, instead of showing it only per VM.
 - Add an ssh key during template creation to be used for Ansible, instead of using username and password.
 - Keep track of existing DHCP-servers and only offer the option to deploy a new one when the selected VLAN ID doesn't already contain a DHCP server.

Conclusion

We have delivered a project that simplifies the deployment of virtual lab environments. As it is a standalone project that is fairly easy to set up and does not depend on a CI/CD pipeline, it is especially suitable for smaller environments. It is also easy to extend, for example adding support for AWS instead of vSphere should be as simple as grabbing a new Terraform template and copying the Ansible provider part into that.

Looking back on the goals, we have achieved them in the following ways:

- **Built for small environments:** This project is fairly easy to get up and running, so we can consider this goal to be satisfied.
- **No CI/CD dependency:** In relation with the goal mentioned above, the project does not depend on a CI/CD pipeline and it is fully standalone, so this goal is satisfied as well.
- **Modular:** As the project has three distinct stages, it is possible to replace the individual tools used for this, provided they offer similar functionality. For example it is easy to swap Packer for another tool that can build templates, as the second script does not depend on Packer itself. We consider this goal to be satisfied as well.
- **Easy to extend:** The IaC CLI offers an easy way to execute custom Ansible playbooks during the configuration stage, and adding new playbooks is as easy as copying them into the ansible directory. There is no user-friendly way yet to execute custom scripts during the template building stage, however this should be as easy as adding a line to the `{distro}.pkr.hcl` file. We consider this goal mostly satisfied.
- **Security of credentials:** The project currently only uses Vault in the second script, and as such, Packer expects the credentials to be in the configuration files. As mentioned above, this is an area for future work, so we consider this goal somewhat satisfied.

All in all, the IaC CLI is a much easier and more flexible, secure and user friendly way to build and deploy new lab environments than doing it manually.