Automation Demo

This is a small demo that will show the concepts of application automation

Objectives

- Discuss and explain issues of authentication and access with automation
- Understand secrets management and discuss potential software solutions
- Discuss and understand the creation and use of SSH/RSA keys in relation to private repositories
- Describe and Understand the functional domains of Packer and Vagrant
  - How to manage the creation of virtual machine images/artifacts with Packer
  - How to manage, create, and start those Virtual Machines in Vagrant
  - How to manage private networking in VirtualBox/Vagrant for application deployment
- Describe and understand the concept of provisioning and post-processing of virtual machine artifacts

Outcomes

At the end of this demonstration you will be able to deploy create two virtual machines, containing a sample NodeJS application and a sample relational database, via Packer and run/managed via Vagrant. You will have successfully deployed this application code from a private GitHub repo via SSH and securely used RSA keys and Linux ENV variables to do so.

Setup

For this demonstration, you need to have Packer, Vagrant, and VirtualBox installed. If you completed the Tooling Assignment then you will have the correct tools.

You can clone the sample code I will be using–issue the command:

```
git clone https://github.com/jhajek/packer-vagrant-build-scripts.git
```

```
# The sample application code is located in my public repo jhajek
# you can clone this and copy it to your own private repo, located in the itmt-430 directory > sample code
git clone https://github/com/illinoistech-itm/jhajek.git
```

In this repo, under the folder named **Packer > itmt430 > automation-demo** you will find a detailed Readme.md. The content of that link will be reprinted here.

Initial Problem

How can you clone code from a private GitHub Repo? When you do so on the command line you are prompted for a password. This breaks automation. Git and GitHub have a solution. You can add an RSA Public Key to GitHub (called a Deploy Key) and clone over SSH as opposed to https.

Pre-steps

Secrets management (passwords, keys, key/value pairs) is the one of the main focuses of cyber-security as well as any application health. In building an application via an automation tools, we have to consider how we will place secrets into our new Operating System as well as how we will retrieve application code securely from a private GitHub repository.

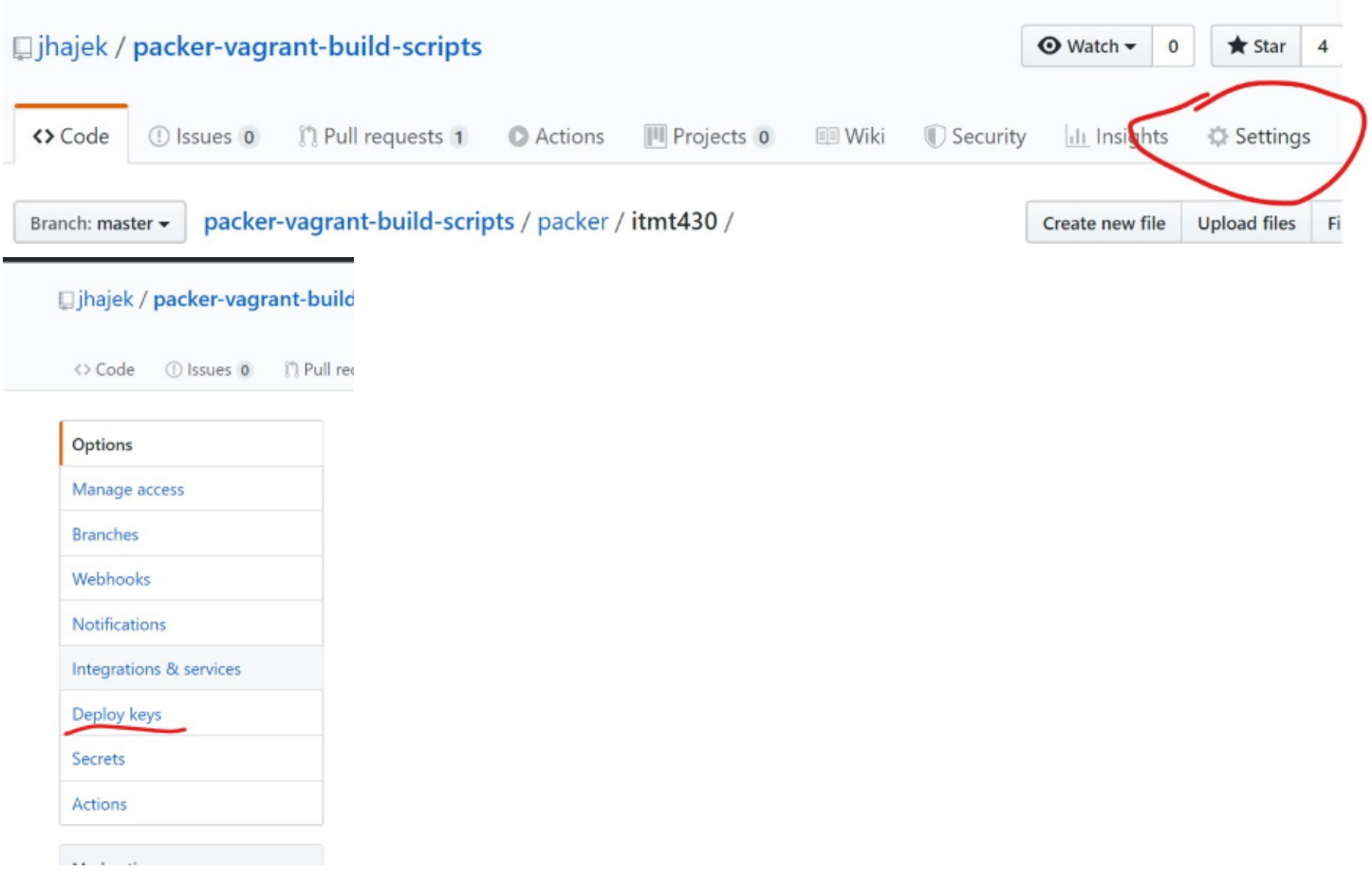
We will be using SSH/RSA keys to authenticate between your system and GitHub. You might have heard these referred to as public/private key authentication. You will need to generate a keypair. This can be done by issuing the following command from a shell on Mac, Linux, and/or PowerShell/Windows:

```
# this is the command to generate a basic keypair
ssh-keygen
```

You can accept the defaults by hitting enter. In this case we don't want a passcode for our key. If you have not generated a keypair before two files, `id_rsa` and `id_rsa.pub` will be generated in the `~/.ssh` directory. One is the private key, one is the public key.

Deploy Keys

Upon being made an admin of your team's GitHub repo, you will now have the ability to add public keys to your team private repo. GitHub refers to these as **Deploy Keys**.



From the command line display the content of your Public Key, copy and paste the content into a new GitHub Deploy Key. Give the Deploy Key name a very descriptive name so you can remember where the key is located. You don't want to be reusing keys or passing them around. You can generate and add as many keys as you want to. Each person should have your own generated key.

```
controller@lenovo-laptop:~/.ssh$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDKdv3b35jdgr0YclLhh1PLZMWw5LHUwSvyZJ6ntet+XQqq1bbY2/jG7NCJYwVXFyuAGOuHA4CaDNJCR3MP
qoxe+6b5EigoNQIDQTbZNreOroVykGYuaL5/V7qEuqleMAeQSLU7lsUjLSeeI92rVAC2fu8bHKxa+zoDrvBMxzWb4FVNdyercgEsJAhR/NjgYpsRZfmFsT89
qZ2pihsBrICgQNT1pvgyjRXkAugTTttbNpeeIcWuajTIi09PxZmjPclB9wC5KTU702fkQDbbBZ93QneaPUiG1bhOQjYJcfBKTfsGHm7uJdtJ7kkEqP3pQwdL
KuOie6oC6tXBWYDeO47j controller@lenovo-laptop
controller@lenovo-laptop:~/.ssh$
```

Figure 1: *id_rsa.pub*

SSH Config

What happens when we run this command?

```
# replace hajek.git with your teams or your own private repo name
git clone git@github.com:illinoistech-itm/hajek.git
```

You are prompted with a yes/no prompt, which will prevent an automated cloning.

There is a way to disable the fingerprinting. It involves creating a file called `config` and placing it in your `~/.ssh` directory. This `config` file is an SSH default file that any SSH connection will read automatically. This helps setup FQDNs or complicated file paths and saves you the time to type them. The file should contain this content at the least:

```
Host   github.com
Hostname github.com
IdentityFile  /home/vagrant/.ssh/id_rsa
StrictHostKeyChecking no
```

```
controller@lenovo-laptop:~/.ssh$ git clone git@github.com:illinoistech-itm/hajek.git
Cloning into 'hajek'...
The authenticity of host 'github.com (140.82.113.4)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)?
```

Figure 2: *Yes/No fingerprinting*

- The value `StrictHostKeyChecking` is what turns off the fingerprint checking.
- The `IdentityFile` value should point to your private key location (id_rsa)

After adding the `config` file, we now encounter a file permission error. You will see the issue here is that GitHub requires private key files file to be permission 600 if you are on Linux or Mac, but not Windows.

```
controller@lenovo-laptop:~$ git clone git@github.com:illinoistech-itm/hajek.git
Cloning into 'hajek'...
Bad owner or permissions on /home/controller/.ssh/config
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
controller@lenovo-laptop:~$ ls -la ~/.ssh/config
-rw-rw-rw- 1 controller controller 106 Feb 27 10:35 /home/controller/.ssh/config
controller@lenovo-laptop:~$ chmod 600 ~/.ssh/config
controller@lenovo-laptop:~$ git clone git@github.com:illinoistech-itm/hajek.git
Cloning into 'hajek'...
Warning: Permanently added 'github.com,192.30.253.112' (RSA) to the list of known hosts.
remote: Enumerating objects: 319, done.
remote: Counting objects: 100% (319/319), done.
remote: Compressing objects: 100% (253/253), done.
remote: Total 6169 (delta 189), reused 181 (delta 58), pack-reused 5850
Receiving objects: 100% (6169/6169), 185.75 MiB | 6.20 MiB/s, done.
Resolving deltas: 100% (3724/3724), done.
Checking out files: 100% (674/674), done.
controller@lenovo-laptop:~$
```

Figure 3: *bad-permission*

A word about automation

Looking back to the dawn of the PC era and the operating systems that flowed from that era, Windows, MacOS, and Linux, we see that installation of these OSes are tailored for single person installs with many manual options and configurations. As students we learned this in ITM 301 or a related Intro to Computers class, where we install many operating systems.

Outside of a technology called Jumpstart "Sun Solaris Jumpstart network based install tool wike page") which was successful doing network based installs, the idea deploying operating systems in an automated fashion was difficult, as operating systems were not built for this kind of thing.

Coming into the current decade (2010), we begin to see the prevalence of a few things;

- High speed and Wireless Internet
- Laptops comparable in memory, disk, and CPU to desktops
- Massive increase in hard disk capacity and in speeds (SSD, NVMe)
- Virtualization – through VirtualBox being opensource software
  - Virtual networking and routing through the VirtualBox Network interface
- Elastic Self-Serve Cloud Computing

With this combination, the idea of building a network and multiple machines that represents your application became possible. The remainder of this tutorial will show you a way that based on your project this can be achieved. As always these are not the only tools available, but a large part of the industry will be familiar with this method. And always there are further optimizations that I will mention but are not required–feel free to explore.

Packer.io

Packer is an open source tool for creating identical machine images for multiple platforms from a single source configuration. Packer is lightweight, runs on every major operating system, and is highly performant, creating machine images for multiple platforms in parallel...

A machine image is a single static unit that contains a pre-configured operating system and installed software which is used to quickly create new running machines. Machine image formats change for each platform. Some examples include AMIs for EC2, VMDK/VMX files for VMware, OVF exports for VirtualBox, etc.

Why Use Packer

The verbiage is taken from Why Use Packer?

Pre-baked machine images have a lot of advantages, but most have been unable to benefit from them because images have been too tedious to create and manage. There were either no existing tools to automate the creation of machine images or they had too high of a learning curve. The result is that, prior to Packer, creating machine images threatened the agility of operations teams, and therefore aren't used, despite the massive benefits.

Packer changes all of this. Packer is easy to use and automates the creation of any type of machine image. It embraces modern configuration management by encouraging you to use a framework such as Chef or Puppet to install and configure the software within your Packer-made images.

In other words: Packer brings pre-baked images into the modern age, unlocking untapped potential and opening new opportunities.

Advantages of Using Packer

Super fast infrastructure deployment. Packer images allow you to launch completely provisioned and configured machines in seconds, rather than several minutes or hours. This benefits not only production, but development as well, since development virtual machines can also be launched in seconds, without waiting for a typically much longer provisioning time.

- Multi-provider portability
  - Because Packer creates identical images for multiple platforms, you can run production in AWS, staging/QA in a private cloud like OpenStack, and development in desktop virtualization solutions such as VMware or VirtualBox. Each environment is running an identical machine image, giving ultimate portability.
- Improved stability
  - Packer installs and configures all the software for a machine at the time the image is built. If there are bugs in these scripts, they'll be caught early, rather than several minutes after a machine is launched.
- Greater testability
  - After a machine image is built, that machine image can be quickly launched and smoke tested to verify that things appear to be working. If they are, you can be confident that any other machines launched from that image will function properly.

Packer makes it extremely easy to take advantage of all these benefits.

Packer Structure

Packer uses JSON based build template files to provide all the answers needed for an automated installation. You can find many working samples all over the internet, but I have provided 2 bare-bones or vanilla samples using CentOS Server and Ubuntu Server. You can clone the repo for these samples here:

```
# public repo containing Packer.io working build template samples.   The samples are located in the directory:
git clone https://github.com/jhajek/packer-vagrant-build-scripts.git
```

Let us take a look at the file: `ubuntu-18044-vanilla.json`. It is located here for viewing purposes.

If you are not familiar with JSON, it is a way to create key-value paired *objects* in basic text format, in order to give text data a searchable format, as well as make data-interchange possible between languages.

The build template file, is this case named `ubuntu-18044-vanilla.json` has three major sections:

- builders
  - The builder section tells packer what you are going to be building – such as an VirtualBox VM or a Hyper-V VM for instance. The values in this section the would relate to the options you would select if you were to manually create a new VirtualBox virtual machine.
- provisioners
  - This section is the added benefit to Packer. This step executes after the base install is finished.
  - This is where you can secure copy SSH keys into the new virtual machine, and/or execute shell scripts to add additional software or configure your software.
- post-processors
  - This phase is once the builder artifact is completed, you can export that format into multiple other formats.
  - This allows you to create a VirtualBox OVF file and then convert it into an Amazon EC2 AMI or in our case we will be converting into a Vagrant Box.

Run the Sample

You can build a vanilla Ubuntu Server 18.04.4 and/or a Centos 19.08 Vagrant Box by running this command from the directory where the build templates (JSON) are located:

```
# command to build Vagrant boxes with Packer
packer build ubuntu18044-vanilla.json
packer build centos-7-vanilla.json
```

These samples contain no code from any GitHub repos so they will simply complete their automated install and leave you with an artifact named: `*.box` located in the `../build` directory

How does it answer install questions

Packer makes use of existing answer file technology. Debian/Ubuntu uses a file format called preseed and RedHat/CentOS uses a file format called **kickstart** which is an homage to Sun's Jumpstart.

For the Ubuntu 18.04.4 template you can see this on line 28 and line 24 in the Centos-7 template. These have been provided for you and will work out of the box ™ without having to modify anything as long as the **ks** or **preseed** directory are located in the same level as the build template.

Output and Vagrant

Once the Packer build template finishes, you will have a *.box file. This conversion step from an OVF to a Vagrant file is specified in the post-processing step, line 42-46 in the Ubuntu sample. The value `output` is the directory of where you would like the artifact to be placed. I choose to make a directory called `../build` relative to my build templates and have all artifacts placed there.

This is a convention that makes sense to me, but you can change this to match your own convention if you desire.

The `*.box` artifact is good to have but doesn't do us any good. Vagrant is a virtualization abstraction tool. We need to add our newly created *.box so that Vagrant can manage it. Let's walk through this.

Say you have an artifact named: `ubuntu-ws-18044-server-virtualbox-1583722003.box`. The numbers on yours will be different because I added a system variable that always appends epoch time to the filename guaranteeing a unique filename every time. Packer build will fail if you already have an existing artifact with the same name (won't overwrite unless you use a –force flag).

To see which boxes Vagrant is currently managing, from the commandline type:

```
# Command to see what boxes Vagrant is managing
vagrant box list
```

```
# Command to add the newly built box to Vagrant for management
# Note, the end of your file name will be different as the timestamp will be different
# the --name potion is important as this is how Vagrant will refer this virtual machine
vagrant box add ./ubuntu-ws-18044-server-virtualbox-1583722003.box --name ubuntu-vanilla
vagrant box list
```

You should see your Vagrant box in the list as: ubuntu-vanilla

Vagrant init

The next step is to create a properties file for this virtual machine called a **Vagrantfile**. This is essentially a configuration file that is interpreted on boot time of the virtual machine and the configuration is translated into VirtualBox CLI commands to create the desired configuration.

To create this file, I would strongly suggest creating a directory first with the name of the Vagrant box. This helps keep things clear as to which virtual machine this controls and secondly every Vagrantfile has the same name Vagrantfile, and you wouldn't want to overwrite existing files.

```
# Commands to create a directory and create a Vagrantfile for ubuntu-vanilla
# Assuming that your PWD is a directory called build
mkdir ubuntu-vanllia
cd ubuntu-vanilla
vagrant init ubuntu-vanilla
```

Once these steps are done, you can launch the virtual machine with a simple command:

```
#  Command to launch and connect to a Vagrant Box
# From the CLI with the location the directory where the Vagrantfile is located
vagrant up
# to remotely connect
vagrant ssh
```

Vagrant Private Networking

Vagrant has an added feature that will enable a local routable private network on your host PC between all VMs and your host. It has the added benefit of handling IP address assignment - forging the need to have a DHCP server. This will allow each person to work on their own hardware without the need for a lab or network. Meaning you can work at home, in the school library, in the lab, all without having to change your networking stack.

Located in the Vagrantfile on line ~35 you will find this commented out code block.

```
  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  # config.vm.network "private_network", ip: "192.168.33.10"
```

Uncomment the `config.vm.network` line to automatically assign the listed private IP to your ubuntu-vanilla machine.

```
  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
    config.vm.network "private_network", ip: "192.168.33.10"
```

If you have multiple machines, you would change the IP addresses and you now can have a fully assignable network as well as be able to use FQDNs out of your `/etc/hosts` file. This allows you to add premade Vagrantfiles to your GitHub repo, which you can clone and script the process of starting the VM automatically.

One thing you will have to do in this case is if you have run the `vagrant up` command already and make a change to the `Vagrantfile` you will need to exit the `vagrant ssh` session and issue a `vagrant reload --provision` which is the command that forces Vagrant to re-evaluate the `Vagrantfile` for any changes it needs to make to the structure of the VM. This only needs to take place once. If you change the Vagrantfile.

Application Sample

The vanilla installs are a good start, but they don't answer the question of **secrets** and how to pass them.

What kind of secrets might we have?

- Private keys
- Usernames and Passwords
- Authorization keys
- JWT seeds
- Salts

Packer User-Variables

https://www.packer.io/docs/templates/user-variables.html

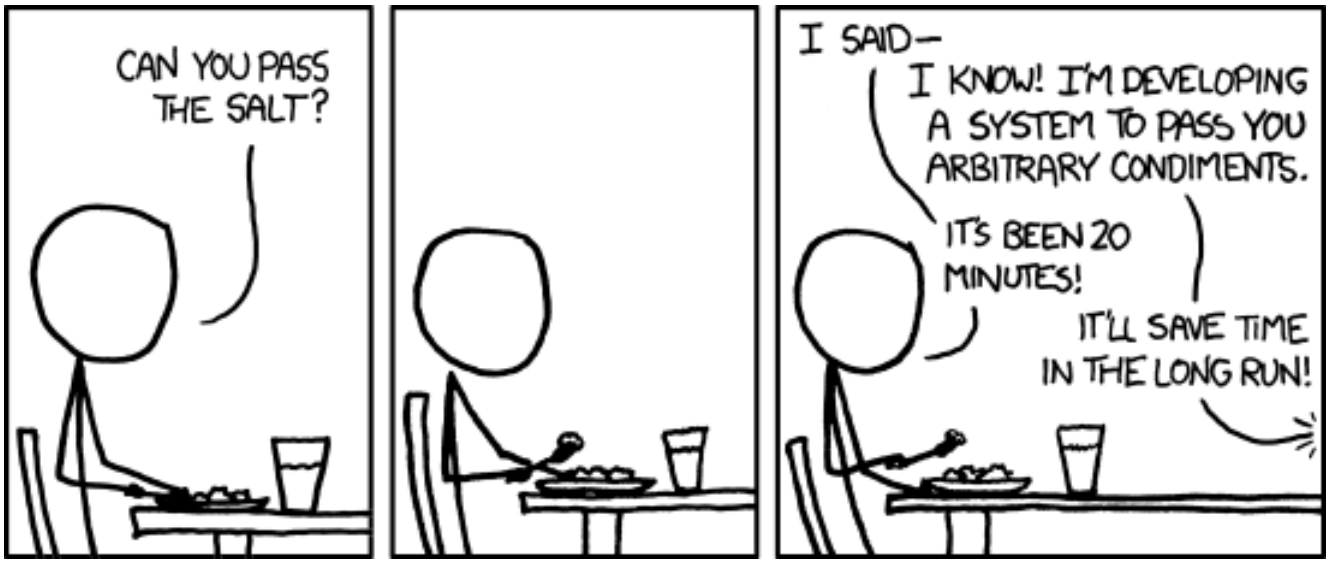


Figure 4: *The General Problem*

In this example we will be using Packer and Vagrant. Packer will be used to construct and automate the build of our application. Packer makes use of `scp` commands during its post-provisioner phase in order to allow files and other secrets to be transferred.

What we need to set username and passwords securely in Packer

1) Issue the command inside of the folder, `cp variables-sample.json variables.json`
    1) The `variables.json` file contains key value pairs of variables and passwords to be passed into the provisioner shell script.
    2) This renames the file `variables-sample.json` to `variables.json` (There is an entry in the `.gitignore` so you cannot accidentally `git push` your passwords).
2) Edit the `variables.json` file replacing default values with your own
3) Issue the command `packer build --var-file=./variables.json ubuntu18044-itmt430-database.json` and `packer build --var-file=./variables.json ubuntu18044-itmt430-webserver.json` to begin the install with password, usernames, and RSA private key properly seeded
4) This way we can securely build the system, deploy it and when building it pass in passwords via environment variables

Webserver contents

1) This application has an Nginx webserver running on port 80.

2) It has a Nodejs Hello World application running on port 3000.
3) It has an Nginx route to the Nodejs app located at `/app`

Database contents

1) System will create a `.my.cnf` file which allows for password-less authentication
2) System will pre-seed MariaDB or MySql root password