Chapter 10: Automated Windows Guest Deployment and Customization

Now that we have covered the different ways of deploying Linux-based **Virtual Machines** (**VMs**) in KVM, it's time to switch our focus to Microsoft Windows. Specifically, we'll work on Windows Server 2019 machines running on KVM, and cover prerequisites and different scenarios for the deployment and customization of Windows Server 2019 VMs. This book isn't based on the idea of **Virtual desktop infrastructure** (**VDI**) and desktop operating systems, which require a completely different scenario, approach, and technical implementation than virtualizing server operating systems.

In this chapter, we will cover the following topics:

- The prerequisites to creating Windows VMs on KVM
- Creating Windows VMs using the **`virt-install`** utility
- The customization of Windows VMs using **`cloudbase-init`**
- **`cloudbase-init`** customization examples
- Troubleshooting common **`cloudbase-init`** customization issues

The prerequisites to creating Windows VMs on KVM

When starting the installation of a guest operating system on KVM virtualization, we always have the same starting points. We need either of the following:

- An ISO file with operating system installation
- An image with a VM template
- An existing VM to clone and reconfigure

Let's start from scratch. We are going to create a Windows Server 2019 VM in this chapter. Version selection was made to keep in touch with the most recent release of Microsoft server operating systems on the market. Our goal will be to deploy a Windows Server 2019 VM template that we can use later for more deployments and **`cloudbase-init`**, and the tool of choice for this installation process is going to be **`virt-install`**. If you need to install an older version (2016 or 2012), you need to know two facts:

- They are supported on CentOS 8 out of the box.
- The installation procedure is the same as it will be with our Windows Server 2019 VM.

If you want to use Virtual Machine Manager to deploy Windows Server 2019, make sure that you configure the VM properly. That includes selecting the correct ISO file for the guest operating

system installation, and connecting another virtual CD-ROM for **`virtio-win`** drivers so that you can install them during the installation process. Make sure that your VM has enough disk space on the local KVM host (60 GB+ is recommended), and that it has enough horsepower to run. Start with two virtual CPUs and 4 GB of memory, as this can easily be changed later.

The next step in our scenario is to create a Windows VM that we'll use throughout this chapter to customize via **`cloudbase-init`**. In a real production environment, we need to do as much configuration in it as possible – driver installation, Windows updates, commonly used applications, and so on. So, let's do that first.

Creating Windows VMs using the virt-install utility

The first thing that we need to do is to make sure we have the **`virtio-win`** drivers ready for installation – the VM will not work properly without

them installed. So, let's first install that and the **libguestfs** packages, in case you don't have them already installed on your server:

```
yum –y install virtio-win libguestfs*
```

Then, it's time to start deploying our VM. Here are our settings:

- The Windows Server 2019 ISO is located at **/iso/windows-server-2019.iso**.
- The **virtio-win** ISO file is located in the default system folder, **/usr/share/virtio-win/virtio-win.iso**.
- We are going to create a 60 GB virtual disk, located at the default system folder, **/var/lib/libvirt/images**.

Now, let's start the installation process:

```
virt-install --name WS2019 --
memory=4096 --vcpus 2 --cpu host --
video qxl --
features=hyperv_relaxed=on,hyperv_spi
nlocks=on,hyperv_vapic=on --clock
```

```
  hypervclock_present=yes --disk
  /var/lib/libvirt/images/WS2019.qcow2,
  format=qcow2,bus=virtio,cache=none,si
  ze=60 --cdrom /iso/windows-server-
  2019.iso --disk /usr/share/virtio-
  win/virtio-win.iso,device=cdrom --vnc
  --os-type=windows --os-
  variant=win2k19 --accelerate --noapic
```

When the installation process starts, we have to click **Next** a couple of times before we reach the configuration screen where we can select the disk where we want to install our guest operating system. On the bottom of that screen to the left, there's a button called **Load driver**, which we can now use, repeatedly, to install all of the necessary **`virtio-win`** drivers. Make sure that you untick the **Hide drivers that aren't compatible with this computer's hardware** checkbox. Then, add the following drivers one by one, from a specified directory, and select them with your mouse:

- **`AMD64\2k19`**: **Red Hat VirtIO SCSI controller**.

- **`Balloon\2k19\amd64`: VirtIO balloon driver.**
- **`NetKVM\2k19\AMD64`: Red Hat VirtIO Ethernet adapter.**
- **`qemufwcfg\2k19\amd64`: QEMU FWCfg device.**
- **`qemupciserial\2k19\amd64`: QEMU Serial PCI card.**
- **`vioinput\2k19\amd64`: VirtIO input driver** and **VirtIO input driver helper**; select both of them.
- **`viorng\2k19\amd64`: VirtIO RNG device.**
- **`vioscsi\2k19\amd64`: Red Hat VirtIO SCSI pass-through controller.**
- **`vioserial\2k19\amd64`: VirtIO serial driver.**
- **`viostor\2k19\amd64`: Red Hat VirtIO SCSI controller.**

After that, click **Next** and wait for the installation process to finish.

You might be asking yourself: *why did we micromanage this so early in the installation process, when we could've done this later?* The answer is

two-fold – if we did it later, we'd have the following problems:

- There's a chance – at least for some operating systems – that we won't have all the necessary drivers loaded before the installation starts, which might mean that the installation will crash.
- We'd have loads of yellow exclamation marks in **Device Manager**, which is usually annoying to people.

Being as it is after deployment, our device manager is happy and the installation was a success:

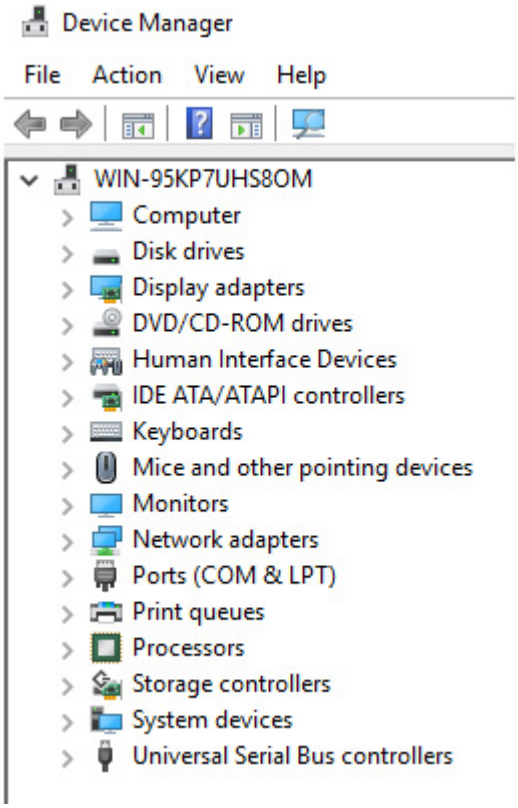

Figure 10.1 – The operating system and all drivers installed from the get-go

The only thing that's highly recommended post installation is that we install the guest agent from **`virtio-win.iso`** after we boot our VM. You will find an **`.exe`** file on the virtual CD-ROM, in **`guest-agent`** directory, and you just need to click the **Next** button until the installation is complete.

Now that our VM is ready, we need to start thinking about customization. Specifically, large-scale

customization, which is a normal usage model for VM deployments in the cloud. This is why we need to use **`cloudbase-init`**, which is our next step.

Customizing Windows VMs using cloudbase-init

If you had the chance to go through ***Chapter 9***, *Customizing a Virtual Machine with cloud-init*, we discussed a tool called **`cloud-init`**. What we used it for was guest operating system customization, specifically for Linux machines. **`cloud-init`** is heavily used in Linux-based environments, specifically in Linux-based clouds, to perform initialization and configuration of cloud VMs.

The idea behind **`cloudbase-init`** is the same, but it's aimed at Windows guest operating systems. Its base services start up when we boot a Windows guest operating system instance, as well as read through the configuration informa-

tion and configure/initialize it. We are going to show a couple of examples of **`cloudbase-init`** operations a bit later in this chapter.

What can **`cloudbase-init`** do? The list of features is quite long, as **`cloudbase-init`** has a modular approach at its core, so it offers many plugins and interpreters, which can be used to further its reach:

- It can execute custom commands and scripts, most commonly coded in PowerShell, although regular CMD scripts are also supported.
- It can work with PowerShell remoting and the **Windows Remote Management (WinRM)** service.
- It can manage and configure disks, for example, to do a volume expansion.
- It can do basic administration, including the following:
  a) Creating users and passwords
  b) Setting up a hostname
  c) Configuring static networking
  d) Configuring MTU size

e) Assigning a license

f) Working with public keys

g) Synchronizing clocks

We mentioned earlier that our Windows Server 2019 VM is going to be used for **`cloudbase-init`** customization, so that's our next subject. Let's prepare our VM for **`cloudbase-init`**. We are going to achieve that by downloading the **`cloudbase-init`** installer and installing it. We can find the **`cloudbase-init`** installer by pointing our internet browser at **https://cloudbase-init.readthedocs.io/en/latest/intro.html#download**. The installation is simple enough, and it can work both in a regular, GUI fashion and silently. If you're used to using Windows Server Core or prefer silent installation, you can use the MSI installer for silent installation by using the following command:

```
msiexec /i CloudbaseInitSetup.msi /qn
/l*v log.txt
```

Make sure that you check the **`cloudbase-init`** documentation for further configuration options

as the installer supports additional runtime options. It's located at **https://cloudbase-init.readthedocs.io/en/latest/**.

Let's stick with the GUI installer as it's simpler to use, especially for a first-time user. First, the installer is going to ask about the license agreement and installation location – just the usual stuff. Then, we're going to get the following options screen:

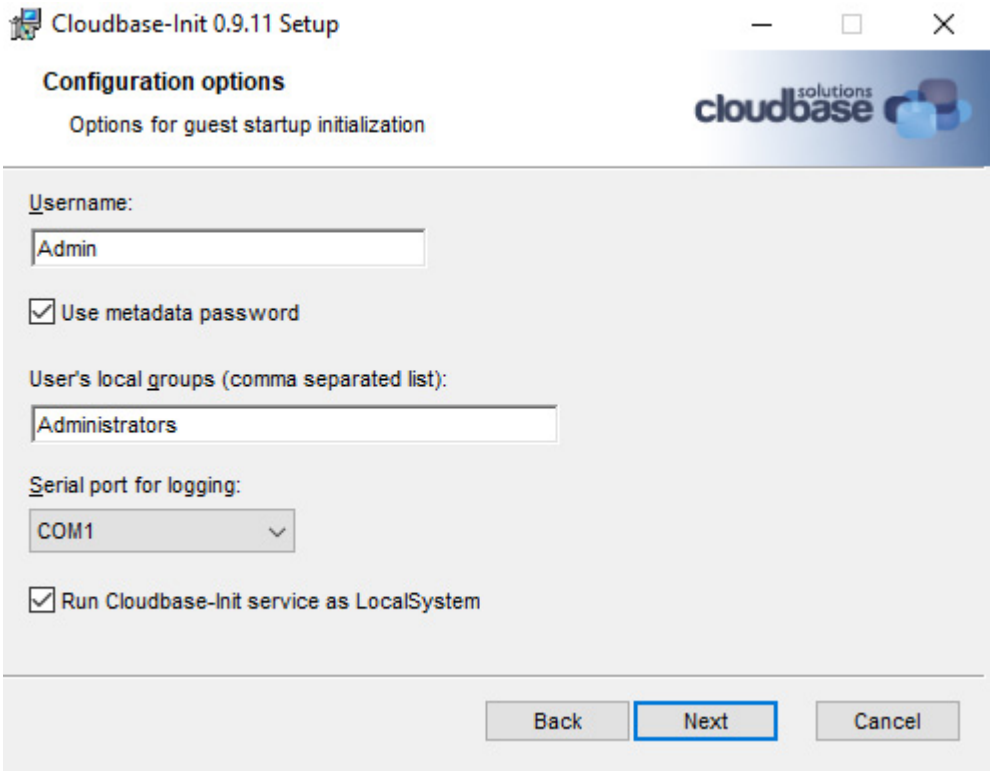

Figure 10.2 – Basic configuration screen

What it's asking us to do is to give permission to create the **`cloudbase-init`** configuration files

(both **`cloudbase-init-unattend.conf`** and **`cloudbase-init.conf`**) with this specific future user in mind. This user will be a member of the local **`Administrators`** group and will be used for logging in when we start using the new image. This will be reflected in both of our configuration files, so if we select **`Admin`** here, that's the user that's going to get created. It's also asking us whether we want the **`cloudbase-init`** service to be run as a **`LocalSystem`** service, which we selected to make the whole process easier. The reason is really simple – this is the highest level of permission that we can give to our **`cloudbase-init`** services so that they can execute its operations. Translation: the **`cloudbase-init`** service will then be run as a **`LocalSystem`** service account, which has unlimited access to all local system resources.

The last configuration screen is going to ask us about running sysprep. Usually, we don't check the **Run sysprep to create a generalized image** box here, as we first need to create a **`cloudbase-`**

init customization file and run sysprep after that. So, leave the following window open:

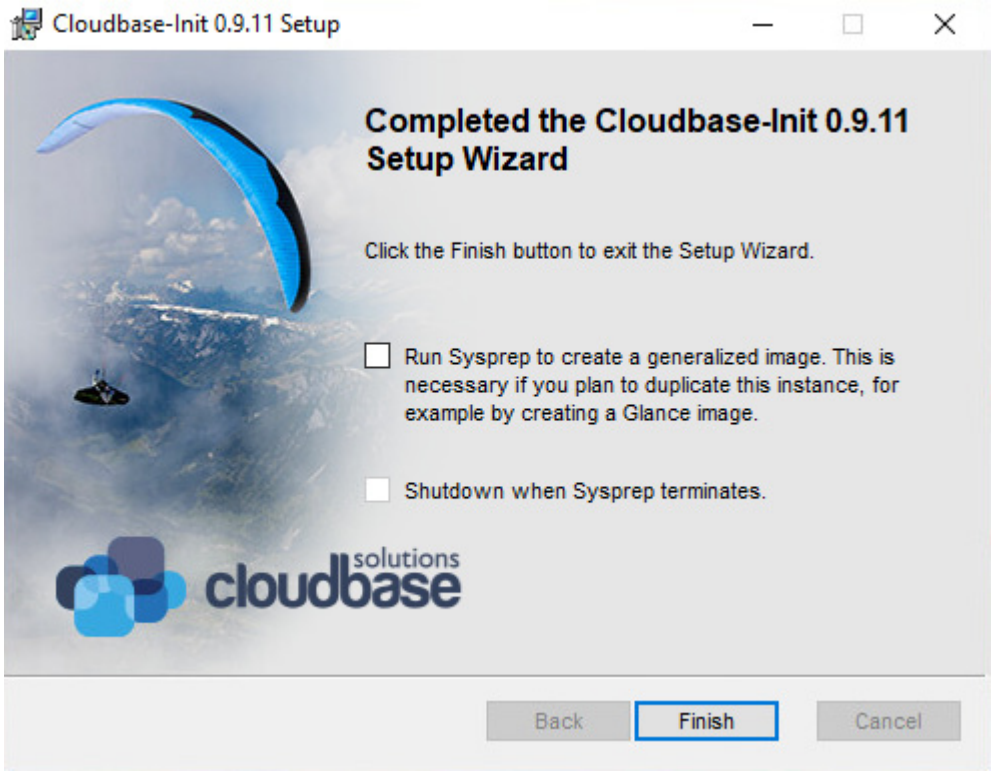

Figure 10.3 – The cloudbase-init installation wizard finishing up

Now that the **cloudbase-init** services are installed and configured, let's create a customization file that's going to configure this VM by using **cloudbase-init**. Again, make sure that this configuration screen is left open (with the completed setup wizard) so that we can easily start the whole process when we finish creating our **cloudbase-init** configuration.

cloudbase-init customization examples

After we finish the installation process, a directory with a set of files gets created in our installation location. For example, in our VM, a directory called **`c:\Program Files\Cloudbase Solutions\Cloudbase-init\`** was created, and it has the following set of subdirectories:

- **`bin`**: The location where some of the binary files are installed, such as **`elevate`**, **`bsdtar`**, **`mcopy`**, **`mdir`**, and so on.
- **`conf`**: The location of three main configuration files that we're going to work with, which is discussed a bit later.
- **`LocalScripts`**: The default location for PowerShell and similar scripts that we want to run post-boot.
- **`Log`**: The location where we'll store the **`cloudbase-init`** log files by default so that we can debug any issues.

- **`Python`**: The location where local installation of Python is deployed so that we can also use Python for scripting.

Let's focus on the **`conf`** directory, which contains our configuration files:

- **`cloudbase-init.conf`**
- **`cloudbase-init-unattend.conf`**
- **`unattend.xml`**

The way that **`cloudbase-init`** works is rather simple – it uses the **`unattend.xml`** file during the Windows sysprep phase to execute **`cloudbase-init`** with the **`cloudbase-init-unattend.conf`** configuration file. The default **`cloudbase-init-unattend.conf`** configuration file is easily readable, and we can use the example provided by the **`cloudbase-init`** project with the default configuration file explained step by step:

```
[DEFAULT]
# Name of the user that will get
created, group for that user
```

```
username=Admin
groups=Administrators
firstlogonbehaviour=no
inject_user_password=true  # Use
password from the metadata (not
random).
```

The next part of the config file is about devices – specifically, which devices to inspect for a possible configuration drive (metadata):

```
config_drive_raw_hhd=true
config_drive_cdrom=true
# Path to tar implementation from
Ubuntu.
bsdtar_path=C:\Program
Files\Cloudbase Solutions\Cloudbase-
Init\bin\bsdtar.exe
mtools_path= C:\Program
Files\Cloudbase Solutions\Cloudbase-
Init\bin\
```

We need to configure some settings for logging purposes as well:

```
# Logging level
```

```
verbose=true
debug=true
# Where to store logs
logdir=C:\Program Files
(x86)\Cloudbase Solutions\Cloudbase-
Init\log\
logfile=cloudbase-init-unattend.log
default_log_levels=comtypes=INFO,suds
=INFO,iso8601=WARN
logging_serial_port_settings=
```

The next part of the configuration file is about networking, so we'll use DHCP to get all the networking settings in our example:

```
# Use DHCP to get all network and NTP
settings
mtu_use_dhcp_config=true
ntp_use_dhcp_config=true
```

We need to configure the location where the scripts are residing, the same scripts that we can use as a part of the **`cloudbase-init`** process:

```
# Location of scripts to be started
during the process
```

```
local_scripts_path=C:\Program
Files\Cloudbase Solutions\Cloudbase-
Init\LocalScripts\
```

The last part of the configuration file is about the services and plugins to be loaded, along with some global settings, such as whether to allow the **cloudbase-init** service to reboot the system or not and how we're going to approach the **cloudbase-init** shutdown process (**false=graceful service shutdown**):

```
# Services for loading
metadata_services=cloudbaseinit.metad
ata.services.configdrive.ConfigDriveS
ervice,
cloudbaseinit.metadata.services.https
ervice.HttpService,
cloudbaseinit.metadata.services.ec2se
rvice.EC2Service,
cloudbaseinit.metadata.services.maass
ervice.MaaSHttpService
# Plugins to load
```

```
plugins=cloudbaseinit.plugins.common.
mtu.MTUPlugin,
        cloudbaseinit.plugins.common.
sethostname.SetHostNamePlugin
# Miscellaneous.
allow_reboot=false    # allow the
service to reboot the system
stop_service_on_exit=false
```

Let's just get a couple of things out of the way from the get-go. Default configuration files already contain some settings that were deprecated, as you're going to find out soon enough. Specifically, settings such as **`verbose`**, **`logdir`** and **`logfile`** are already deprecated in this release, as you can see from the following screenshot, where **`cloudbase-init`** is complaining about those very options:

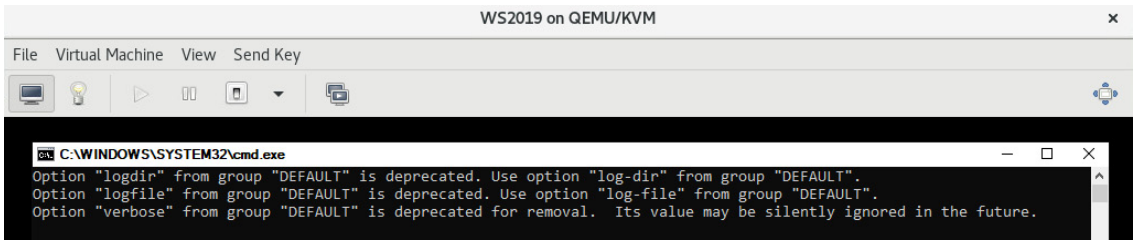

Figure 10.4 – cloudbase-init complaining about its own default configuration file options

If we want to start sysprepping with **`cloudbase-init`** by using the default configuration files, we are actually going to get a pretty nicely configured VM – it's going to be sysprepped, it's going to reset the administrator password and ask us to change it with the first login, and remove the existing administrator user and its directories. So, before we do this, we need to make sure that we save all of our administrator user settings and data (documents, installers, downloads, and so on) someplace safe. Also, the default configuration files will not reboot the VM by default, which might confuse you. We need to do a manual restart of the VM so that the whole process can start.

The easiest way to work with both **`cloud-init`** and **`cloudbase-init`** is by writing down a scenario of what needs to be done to the VM as it gets through the initialization process. So, we'll do just that – pick a load of settings that we want to be configured and create a customization file accordingly. Here are our settings:

- We want our VM to ask us to change the password post-sysprep and after the **`cloudbase-init`** process.
- We want our VM to take all of its network settings (the IP address, netmask, gateway, DNS servers, and NTP) from DHCP.
- We want to sysprep the VM so that it's unique to each scenario and policy.

So, let's create a **`cloudbase-init-unattend.conf`** config file that will do this for us. The first part of the configuration file was taken from the default config file:

```
[DEFAULT]
username=Admin
groups=Administrators
inject_user_password=true
config_drive_raw_hhd=true
config_drive_cdrom=true
config_drive_vfat=true
bsdtar_path=C:\Program
Files\Cloudbase Solutions\Cloudbase-
Init\bin\bsdtar.exe
```

```
mtools_path= C:\Program
Files\Cloudbase Solutions\Cloudbase-
Init\bin\
debug=true
default_log_levels=comtypes=INFO,suds
=INFO,iso8601=WARN
logging_serial_port_settings=
mtu_use_dhcp_config=true
ntp_use_dhcp_config=true
```

As we decided to use PowerShell for all of the scripting, we created a separate directory for our PowerShell scripts:

```
local_scripts_path=C:\PS1
```

The rest of the file was also just copied from the default configuration file:

```
metadata_services=cloudbaseinit.metad
ata.services.base.EmptyMetadataServic
e
plugins=cloudbaseinit.plugins.common.
mtu.MTUPlugin,
        cloudbaseinit.plugins.common.
sethostname.SetHostNamePlugin,
```

```
cloudbaseinit.plugins.common.localscr
ipts.LocalScriptsPlugin,cloudbaseinit
.plugins.common.userdata.UserDataPlug
in
allow_reboot=false
stop_service_on_exit=false
```

As for the **`cloudbase-init.conf`** file, the only change that we made was selecting the correct local script path (reasons to be mentioned shortly), as we will use this path in our next example:

```
[DEFAULT]
username=Admin
groups=Administrators
inject_user_password=true
config_drive_raw_hhd=true
config_drive_cdrom=true
config_drive_vfat=true
```

Also, part of our default config file contained paths for **`tar`**, **`mtools`**, and debugging:

```
bsdtar_path=C:\Program
Files\Cloudbase Solutions\Cloudbase-
```

```
Init\bin\bsdtar.exe
mtools_path= C:\Program
Files\Cloudbase Solutions\Cloudbase-
Init\bin\
debug=true
```

This part of the config file was also taken from the default config file, and we only changed **`local_scripts_path`** so that it's set to the directory that we're using to populate with PowerShell scripts:

```
first_logon_behaviour=no
default_log_levels=comtypes=INFO,suds
=INFO,iso8601=WARN
logging_serial_port_settings=
mtu_use_dhcp_config=true
ntp_use_dhcp_config=true
local_scripts_path=C:\PS1
```

We can then go back to the **`cloudbase-init`** installation screen, check the sysprep option, and click **Finish**. After starting the sysprep process and going through with it, this is the end result:

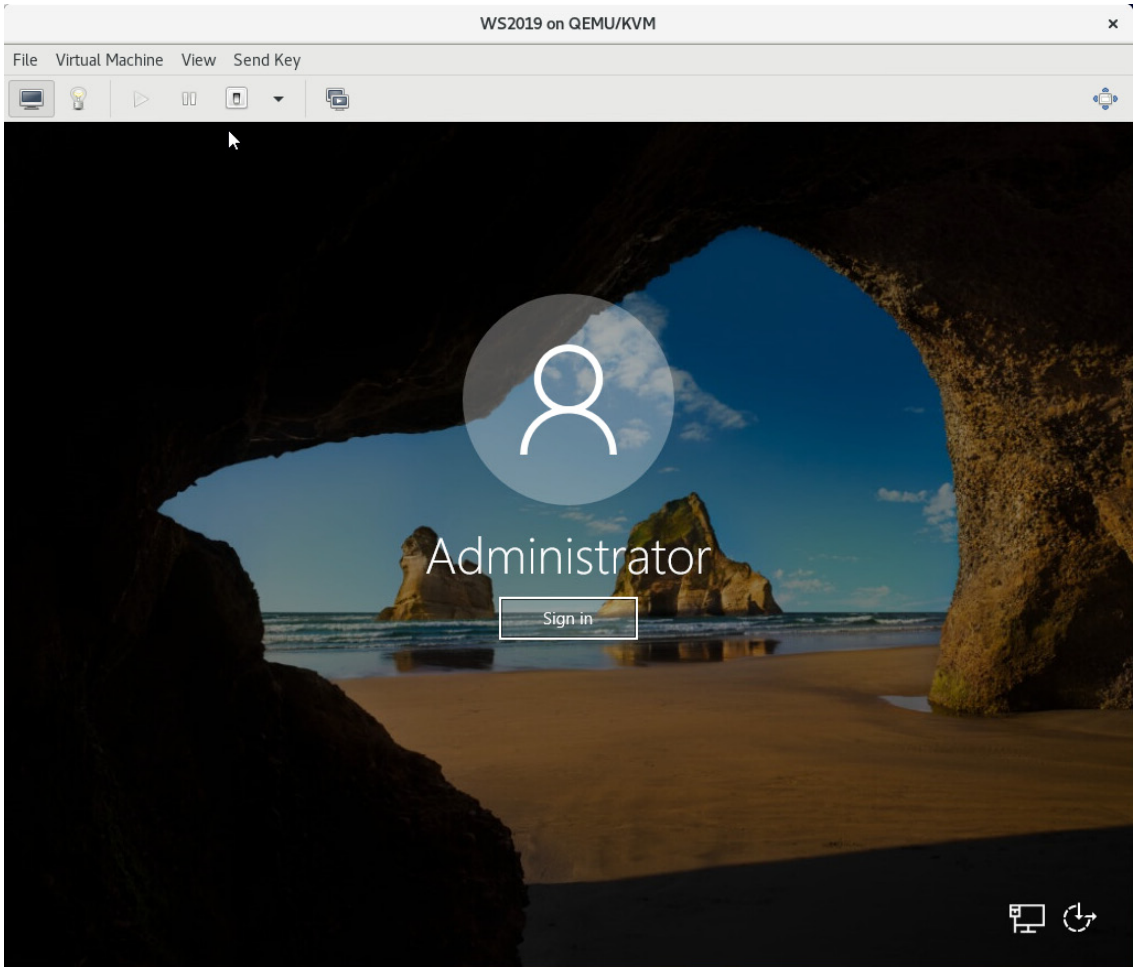

Figure 10.5 – When we press Sign in, we are going to be asked to change the administrator's password

Now, let's take this a step further and complicate things a bit. Let's say that you want to do the same process, but with additional PowerShell code that should do some additional configuration. Consider the following example:

- It should create another two local users called **`packt1`** and **`packt2`**, with a predefined pass-

word set to **Pa$$w0rd**.

- It should create a new local group called **students**, and add **packt1** and **packt2** to this group as members.
- It should set the hostname to **Server1**.

The PowerShell code that enables us to do this should have the following content:

```
Set-ExecutionPolicy -ExecutionPolicy
Unrestricted -Force
$password = "Pa$$w0rd" | ConvertTo-
SecureString -AsPlainText -Force
New-LocalUser -name "packt1" -
Password $password
New-LocalUser -name "packt2" -
Password $password
New-LocalGroup -name "Students"
Add-LocalGroupMember -group
"Students" -Member "packt1","packt2"
Rename-Computer -NewName "Server1" -
Restart
```

Taking a look at the script itself, this is what it does:

- Sets the PowerShell execution policy to unrestricted so that our host doesn't stop our script execution, which it would do by default.
- Creates a password variable from a plaintext string (**`Pa$$w0rd`**), which gets converted to a secure string that we can use with the **`New-LocalUser`** PowerShell cmdlet to create a local user.
- **`New-LocalUser`** is a PowerShell cmdlet that creates a local user. Mandatory parameters include a username and password, which is why we created a secure string.
- **`New-LocalGroup`** is a PowerShell cmdlet that creates a local group.
- **`Add-LocalGroupMember`** is a PowerShell cmdlet that allows us to create a new local group and add members to it.
- **`Rename-Computer`** is a PowerShell cmdlet that changes the hostname of a Windows computer.

We also need to call this code from **`cloudbase-init`** somehow, so we need to add this code as

script. Most commonly, we'll use a directory called **`LocalScripts`** in the **`cloudbase-init`** installation folder for that. Let's call this script **`userdata.ps1`**, save the content mentioned previously to it in the folder, as defined in the **`.conf`** file (**`c:\PS1`**), and add a **`cloudbase-init`** parameter at the top of the file:

```
# ps1
$password = "Pa$$w0rd" | ConvertTo-
SecureString -AsPlainText -Force
New-LocalUser -name "packt1" -
Password $password
New-LocalUser -name "packt2" -
Password $password
New-LocalGroup -name "Students"
Add-LocalGroupMember -group
"Students" -Member "packt1","packt2"
Rename-Computer -NewName "Server1" –
Restart
```

After starting the **`cloudbase-init`** procedure again, which can be achieved by starting the **`cloudbase-init`** installation wizard and going

through it as we did in the previous example, here's the end result in terms of users:

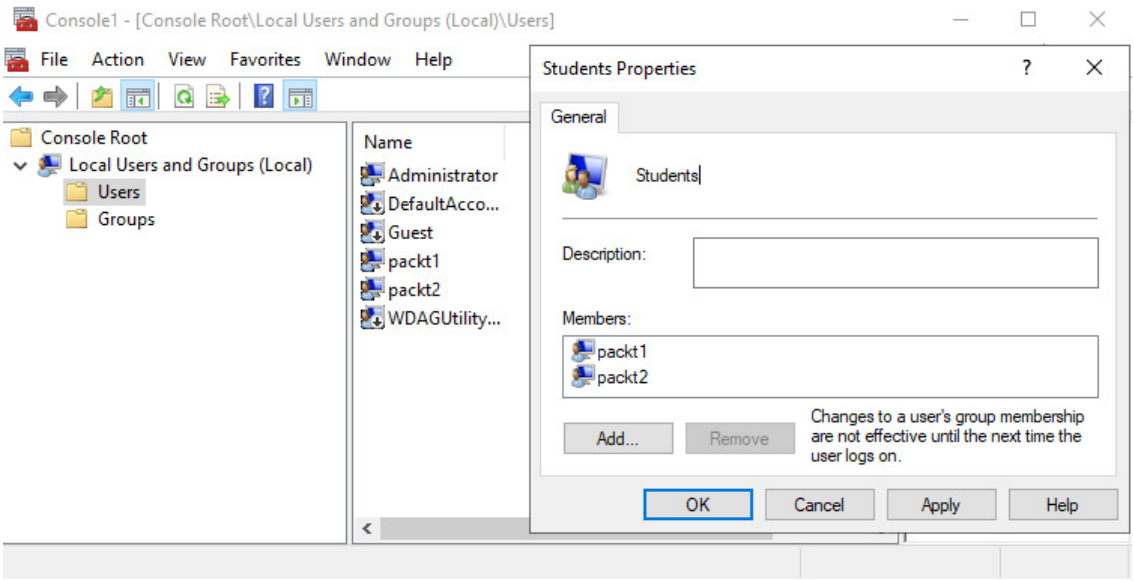

Figure 10.6 – The packt1 and packt2 users were created, and added to the group created by our PowerShell script

We can clearly see that the **`packt1`** and **`packt2`** users were created, along with a group called **`Students`**. We can then see that the **`Students`** group has two members – **`packt1`** and **`packt2`**. Also, in terms of setting the server name, we have the following:

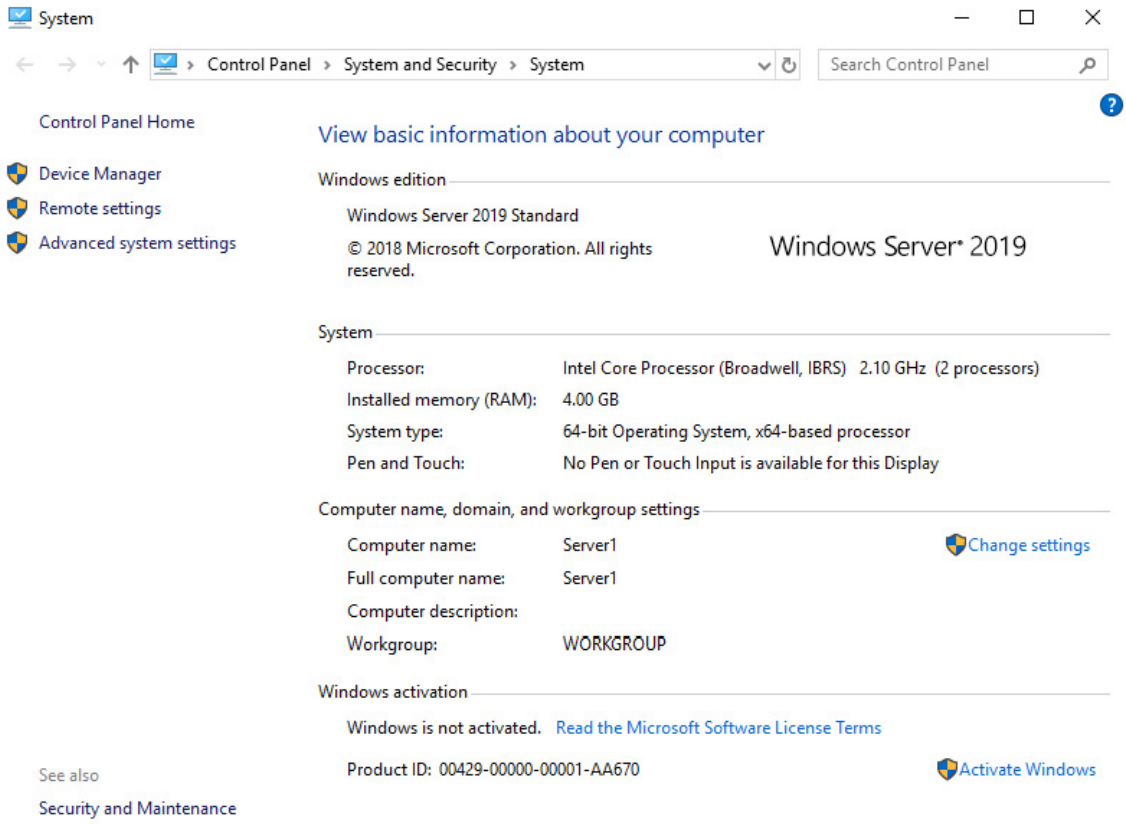

Figure 10.7 – Slika 1. Changing the server name via PowerShell script also works

Using **`cloudbase-init`** really isn't simple, and requires a bit of investment in terms of time and tinkering. But afterward, it will make our job much easier – not being forced to do pedestrian tasks such as these over and over again should be a reward enough, which is why we need to talk a little bit about troubleshooting. We're sure that you'll run into these issues as you ramp up your **`cloudbase-init`** usage.

Troubleshooting common cloudbase-init customization issues

In all honesty, you can freely say that the **`cloud-base-init`** documentation is not all that good. Finding examples of how to execute PowerShell or Python code is difficult at best, while the official page doesn't really offer any help in that respect. So, let's discuss some of the most common errors that happen while using **`cloudbase-init`**.

Although this seems counter-intuitive, we had much more success getting **`cloudbase-init`** to work with the latest development version instead of the latest stable one. We're not exactly sure what the problem is, but the latest development version (at the time of writing, this is version 0.9.12.dev125) worked for us right out of the gate. With version 0.9.11., we had massive issues with getting the PowerShell script to even start.

Apart from these issues, there are other issues that you will surely encounter as you get to know **`cloudbase-init`**. The first one is the reboot loop. This problem is really common, and it almost always happens because of two reasons:

- A mistake in the configuration file – a wrongly typed name of a module or an option, or something like that
- A mistake in some external file (location or syntax) that's being called as an external script to be executed in the **`cloudbase-init`** process

Making a mistake in configuration files is something that happens often, which throws **`cloudbase-init`** into a weird state that ends up like this:

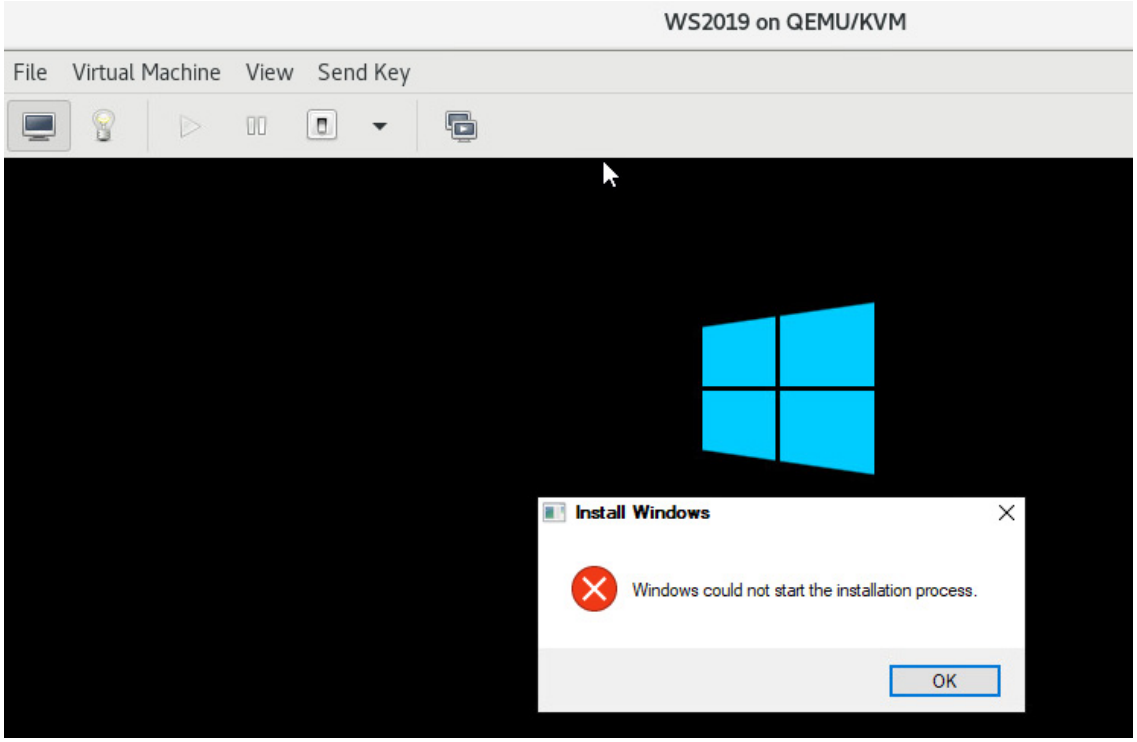

Figure 10.8 – Error in configuration

We've seen this situation multiple times. The real problem is the fact that sometimes it takes hours and hours of waiting, sometimes cycling through numerous reboots, but it's not just a regular reboot loop. It really seems that **`cloudbase-init`** is doing something – the CMD is started, you get no errors in it or on the screen, but it keeps doing something and then finishes like this.

Other issues that you might encounter are even more picky – for example, when **`cloudbase-init`** fails to reset the password during the sysprep/**`cloudbase-init`** process. This can hap-

pen if you manually change the account password that's being used by the **`cloudbase-init`** service (hence, why using **`LocalSystem`** is a better idea). That will lead to the failure of the whole **`cloudbase-init`** procedure, a part of which can be a failure to reset the password.

There's an even more obscure reason why this might happen – sometimes we manually manage system services by using the **`services.msc`** console and we deliberately disable services that we don't immediately recognize. If you set the **`cloudbase-init`** service to be disabled, it will fail in its process, as well. These services need to have automatic startup priority and shouldn't be manually reconfigured to be disabled.

A failure to reset the password can also happen because of some security policies – for example, if the password isn't complex enough. That's why we used a bit more of a complex password in our PowerShell script, as most of us system engineers learned that lesson a long time ago.

Also, sometimes companies have different security policies in place, which can lead to a situation in which some management application that takes care of – for example, software inventory – stops the **`cloudbase-init`** service or completely uninstalls it.

The most frustrating error that we can encounter is an error where the **`cloudbase-init`** process doesn't start scripts from its designated folder. After spending hours perfecting your Python, **`bash`**, cmd, or PowerShell script that needs to be added to the customization process, it's always maddening to see this happen. In order for us to be able to use these scripts, we need to use a specific plugin that is able to call the external script and execute it. That's why we usually use **`UserDataPlugin`** – both for executing and debugging reasons – as it can execute all of these script types and give us an error value, which we can then use for debugging purposes.

One last thing – make sure that you don't insert PowerShell code directly into the **`cloudbase-`**

`init` configuration files in the **`conf`** folder. You'll only get a reboot loop as a reward, so be careful about that.

Summary

In this chapter, we worked with Windows VM customization, a topic that's equally as important as Linux VM customization. Maybe even more so, keeping in mind the market share numbers and the fact that a lot of people are using Windows in cloud environments, as well.

Now that we have covered all the bases in terms of working with VMs, templating, and customization, it's time to introduce a different approach to additional customization that's complementary to **`cloud-init`** and **`cloudbase-init`**. So, the next chapter is about that approach, which is based around Ansible.

Questions

1. Which drivers do we need to install onto Windows guest operating systems so that we can make a Windows template on the KVM hypervisor?
2. Which agent do we need to install onto Windows guest operating systems to have better visibility into the VM's performance data?
3. What is sysprep?
4. What is **`cloudbase-init`** used for?
5. What are the regular use cases for **`cloudbase-init`**?

Further reading

Please refer to the following links for more information:

- Microsoft **`LocalSystem`** account documentation: **https://docs.microsoft.com/en-us/windows/win32/ad/the-localsystem-account**
- **`cloudbase-init`** documentation: **https://cloudbase-**

init.readthedocs.io/en/latest/intro.html

- The **`cloudbase-init`** plugin documentation: **https://cloudbase-init.readthedocs.io/en/latest/plugins.html**
- The **`cloudbase-init`** services documentation: **https://cloudbase-init.readthedocs.io/en/latest/services.html**