

Chapter 16:

Troubleshooting Guidelines for the KVM Platform

If you've followed this book all the way from [***Chapter 1***](#), *Understanding Linux Virtualization*, then you'll know we went through *a lot* together in this book – hundreds and hundreds of pages of concepts and practical aspects, including configuration examples, files and commands – everything. 700 or so pages of it. So far, we've almost completely ignored troubleshooting as part of that journey. We didn't do this on the premise that everything just *works* in Linux and that we didn't have any issues at all and that we achieved a state of *nirvana* while going through this book cover to cover.

It was a journey riddled with various types of issues. Some of them aren't worth mentioning as they were our own mistakes. Mistakes like the ones we made (and you will surely make more too) mostly come from the fact that we mistyped something (in a command or configuration file). Basically, humans play a big role in IT. But some of these issues were rather frustrating. For example, implementing SR-IOV required a lot of time as we had to find different types of problems at the hardware, software, and configuration levels to make it work. oVirt was quite quirky, as we'll soon explain. Eucalyptus was interesting, to put it mildly. Although we used it *a lot* before, cloud-

base-init was really complicated and required a lot of our time and attention, which turned out not to be due to something we did – it was just the cloudbase-init version. But overall, this just further proved a general point from our previous chapter – reading about various IT subjects in books, articles, and blog posts – is a really good approach to configuring a lot of things correctly from the start. But even then, you'll still need a bit of troubleshooting to make everything picture perfect.

Everything is great and amazing once you install a service and start using it, but it seldom happens that way the first time round. Everything we used in this book was actually installed to enable us to test different configurations and grab the necessary screenshots, but at the same time, we wanted to make sure that they can actually be installed and configured in a more structured, procedural way.

So, let's start with some simple things related to services, packages, and logging. Then, we'll move on to more advanced concepts and tools for troubleshooting, described through various examples that we have covered along the way.

In this chapter, we will cover the following topics:

- Verifying the KVM service status
- KVM service logging
- Enabling debug mode logging
- Advanced troubleshooting tools
- Best practices for troubleshooting KVM issues

Verifying the KVM service status

We're starting off with the simplest of all examples – verifying the KVM service status and some of its normal influence on host configuration.

In ***Chapter 3**, Installing a KVM Hypervisor, libvirt, and ovirt*, we did a basic installation of the overall KVM stack by installing **virt module** and using the **dnf** command to deploy various packages. There are a couple of reasons why this might not end up being a good idea:

- A lot of servers, desktops, workstations, and laptops come pre-configured with virtualization turned off in BIOS. If you're using an Intel-based CPU, make sure that you find all the VT-based options and enable them (VT, VT-d, VT I/O). If you're using an AMD-based CPU, make sure that you turn on AMD-V. There's a simple test that you can do to check if virtualization is enabled. If you boot any Linux live distribution, go to the shell and type in the following command:

```
cat /proc/cpuinfo | egrep "vmx|svm"
```

You can also use the following command, if you already installed your Linux host and the appropriate packages that we mentioned in ***Chapter 3**, Installing a KVM hypervisor, libvirt, and ovirt*:

```
virt-host-validate
```

If you don't get any output from this command, your system either doesn't support virtualiza-

tion (less likely) or doesn't have virtualization features turned on. Make sure that you check your BIOS settings.

- Your networking configuration and/or package repository configuration might not be set up correctly. As we'll repeatedly state in this chapter, please, start with the simplest things – don't go off on a journey of trying to find some super complex reason why something isn't working. Keep it simple. For network tests, try using the **ping** command on some well-known server, such as google.com. For repository problems, make sure that you check your **/etc/yum.repos.d** directory. Try using the **yum clean all** and **yum update** commands.

Repository problems are more likely to happen on some other distributions than CentOS/Red Hat, but still, they can happen.

- After the deployment process has finished successfully, make sure that you start and enable KVM services by using the following commands:

```
systemctl enable libvirtd libvirt-  
guests  
systemctl start libvirtd libvirt-  
guests
```

Often, we forget to start and enable the **libvirt-guests** service, and then we get very surprised after we reboot our host. The result of **libvirt-guests** not being enabled is simple. When started, it suspends your virtual machines when you initiate shutdown and resumes them on the next boot. In other words, if you don't enable them, your virtual ma-

chines won't resume after the next reboot.

Also, check out its configuration file,

/etc/sysconfig/libvirt-guests. It's a simple text configuration file that enables you to configure at least three very important settings:

ON_SHUTDOWN, **ON_BOOT**, and **START_DELAY**. Let's explain these:

a) By using the **ON_SHUTDOWN** setting, we can select what happens with the virtual machine when we shut down your host since it accepts values such as **shutdown** and **suspend**.

b) The **ON_BOOT** option does the opposite – it tells **libvirtd** whether it needs to start all the virtual machines on host boot, whatever their autostart settings are. It accepts values such as **start** and **ignore**.

c) The third option, **START_DELAY**, allows you to set a timeout value (in seconds) between multiple virtual machine power-on actions while the host is booting. It accepts numeric values, with **0** being the value for parallel startup and *all other (positive) numbers* being the number of seconds it waits before it starts the next virtual machine.

Considering this, there are at least three things to remember:

- Make sure that these two services are actually running by typing in the following commands:

```
systemctl status libvirtd
systemctl status libvirt-guests
```

At least **libvirtd** needs to be started for us to be able to create or run a KVM virtual machine.

- If you're configuring more advanced settings such as SR-IOV, make sure that you read your server's manual to select a correct slot that is SR-IOV compatible. On top of that, make sure that you have a compatible PCI Express card and BIOS that's configured correctly. Otherwise, you won't be able to make it work.
- When you start the libvirt service, it usually comes with some sort of pre-defined firewall configuration. Keep that in mind in case you ever decide to disable libvirt services as the firewall rules will almost always still be there. That might require a bit of additional configuration.

The next step in your troubleshooting journey will be checking through some of the log files. And there are plenty to choose from – KVM has its own, oVirt has its own, as does Eucalyptus, ELK, and so on. So, make sure that you know these services well so that you can check the correct log files for the situation you're trying to troubleshoot. Let's start with KVM services logging.

KVM services logging

When discussing KVM services logging, there are a couple of locations that we need to be aware of:

- Let's say that you're logged in as root in the GUI and that you started virt-manager. This means that you have a **virt-manager.log** file located in the **/root/.cache/virt-manager di-**

rectory. It's really verbose, so be patient when reading through it.

- The `/etc/libvirt/libvirtd.conf` file is libvirtd's configuration file and contains a lot of interesting options, but some of the most important options are actually located almost at the end of the file and are related to auditing. You can select the commented-out options (`audit_level` and `audit_logging`) to suit your needs.
- The `/var/log/libvirt/qemu` directory contains logs and rotated logs for all of the virtual machines that were ever created on our KVM host.

Also, be sure to check out a command called `au-virt`. It's really handy as it tells you basic information about the virtual machines on your KVM host – both virtual machines that are still there and/or successfully running and the virtual machines that we tried to install and failed at doing so. It pulls its data from audit logs, and you can use it to display information about a specific virtual machine we need as well. It also has a very debug-level option called `--all-events`, if you want to check every single little detail about any virtual machine that was – or still is – an object on the KVM host.

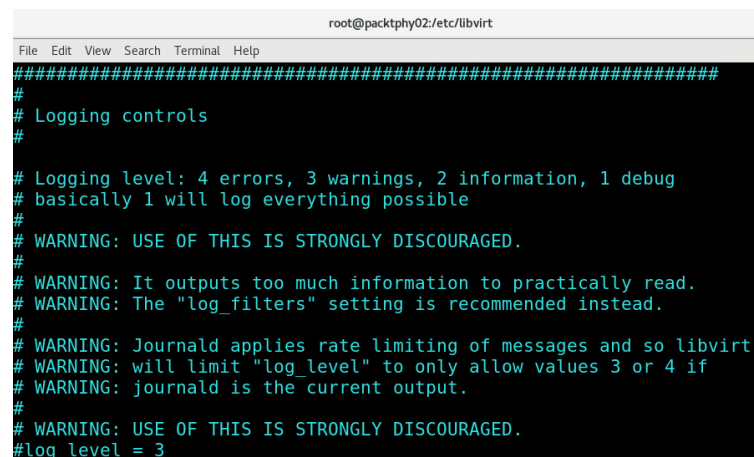
Enabling debug mode logging

There's another approach to logging in KVM: configuring debug logging. In the libvirtd config-

uration file that we just mentioned, there are additional settings you can use to configure this very option. So, if we scroll down to the **Logging controls** part, these are the settings that we can work with:

- **log_level**
- **log_filters**
- **log_outputs**

Let's explain them step by step. The first option – **log_level** – describes log verbosity. This option has been deprecated since libvirt version 4.4.0. In the **Logging controls** section of the file, there's additional documentation hardcoded into the file to make things easier. For this specific option, this is what the documentation says:



```

root@packtphy02:/etc/libvirt
File Edit View Search Terminal Help
#####
#
# Logging controls
#
# Logging level: 4 errors, 3 warnings, 2 information, 1 debug
# basically 1 will log everything possible
#
# WARNING: USE OF THIS IS STRONGLY DISCOURAGED.
#
# WARNING: It outputs too much information to practically read.
# WARNING: The "log_filters" setting is recommended instead.
#
# WARNING: Journald applies rate limiting of messages and so libvirt
# WARNING: will limit "log_level" to only allow values 3 or 4 if
# WARNING: journald is the current output.
#
# WARNING: USE OF THIS IS STRONGLY DISCOURAGED.
#log_level = 3

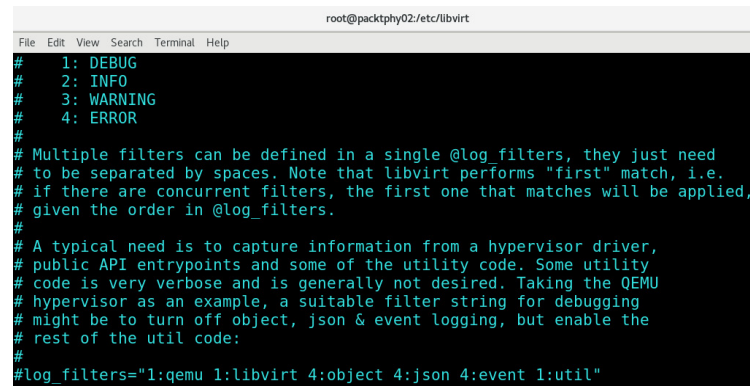
```

Figure 16.1 – Logging controls in libvirtd.conf

What people usually do is see the first part of this output (Logging level description), go to the last line (**log_level**), set it to 1, save, restart the **libvirtd** service, and be done with it. The problem is the text part in-between. It specifically says that **journald** does rate limiting so that it doesn't get hammered with logs from one service

only and instructs us to use the **log_filters** setting instead.

Let's do that, then – let's use **log_filters**. A bit lower in the configuration file, there's a section that looks like this:



```

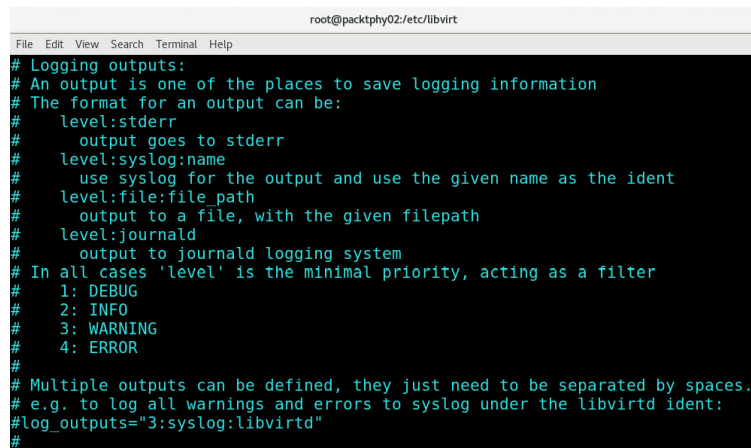
root@packtphy02:/etc/libvirt
File Edit View Search Terminal Help
# 1: DEBUG
# 2: INFO
# 3: WARNING
# 4: ERROR
#
# Multiple filters can be defined in a single @log_filters, they just need
# to be separated by spaces. Note that libvirt performs "first" match, i.e.
# if there are concurrent filters, the first one that matches will be applied,
# given the order in @log_filters.
#
# A typical need is to capture information from a hypervisor driver,
# public API entrypoints and some of the utility code. Some utility
# code is very verbose and is generally not desired. Taking the QEMU
# hypervisor as an example, a suitable filter string for debugging
# might be to turn off object, json & event logging, but enable the
# rest of the util code:
#log_filters="1:qemu 1:libvirt 4:object 4:json 4:event 1:util"

```

Figure 16.2 – Logging filters options in libvirtd.conf

This gives us various options we can use to set different logging options per object types, which is great. It gives us options to increase the verbosity of things that we're interested in at a desired level, while keeping the verbosity of other object types to a minimum. What we need to do is remove the comment part of the last line (**#log_filters="1:qemu 1:libvirt 4:object 4:json 4:event 1:util"** should become **log_filters="1:qemu 1:libvirt 4:object 4:json 4:event 1:util"**) and configure its settings so that they match our requirements.

The third option relates to where we want our debug logging output file to be:



```

root@packtpy02:/etc/libvirt
File Edit View Search Terminal Help
# Logging outputs:
# An output is one of the places to save logging information
# The format for an output can be:
#   level:stderr
#       output goes to stderr
#   level:syslog:name
#       use syslog for the output and use the given name as the ident
#   level:file:file path
#       output to a file, with the given filepath
#   level:journald
#       output to journald logging system
# In all cases 'level' is the minimal priority, acting as a filter
#   1: DEBUG
#   2: INFO
#   3: WARNING
#   4: ERROR
#
# Multiple outputs can be defined, they just need to be separated by spaces.
# e.g. to log all warnings and errors to syslog under the libvirt ident:
#log_outputs="3:syslog:libvirt"
#

```

Figure 16.3 – Logging outputs options in libvirtd.conf

Important Note

*After changing any of these settings, we need to make sure that we restart the **libvirtd** service by typing in the **systemctl restart libvirtd** command.*

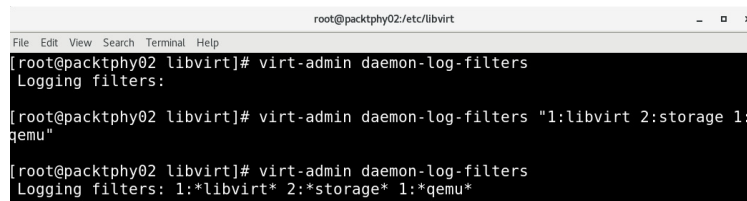
If we're only interested in client logs, we need to set an environment variable called **LIBVIRT_LOG_OUTPUTS** to something like this (let's say we want DEBUG-level logging):

```

export
LIBVIRT_LOG_OUTPUTS="1:file:/var/log/
libvirt_guests.log"

```

All these options are valid until the next **libvirtd** service restart, which is quite handy for permanent settings. However, there's a runtime option that we can use when we need to debug a bit on the fly, without resorting to permanent configuration. That's why we have a command called **virt-admin**. We can use it to set our own settings. For example, let's see how we can use it to get our current settings, and then how to use it to set temporary settings:

A terminal window titled 'root@packtphy02:/etc/libvirt' showing the configuration of logging filters. The user enters 'virt-admin daemon-log-filters' and sees 'Logging filters:'. Then they enter 'virt-admin daemon-log-filters "1:libvirt 2:storage 1:qemu"' and see 'Logging filters: 1:*libvirt* 2:*storage* 1:*qemu*'.

```
root@packtphy02:/etc/libvirt
File Edit View Search Terminal Help
[root@packtphy02 libvirt]# virt-admin daemon-log-filters
Logging filters:
[root@packtphy02 libvirt]# virt-admin daemon-log-filters "1:libvirt 2:storage 1:qemu"
Logging filters: 1:*libvirt* 2:*storage* 1:*qemu*
```

Figure 16.4 – Runtime libvirtd debugging options

We can also delete these settings by issuing the following command:

```
virt-admin daemon-log-filters ""
```

This is something that's definitely recommended after we're done debugging. We don't want to use our log space for nothing.

In terms of straight-up debugging virtual machines – apart from these logging options – we can also use serial console emulation to hook up to the virtual machine console. This is something that we'd do if we can't get access to a virtual machine in any other way, especially if we're not using a GUI in our environments, which is often the case in production environments. Accessing the console can be done as follows:

```
virsh console kvm_domain_name
```

In the preceding command, **kvm_domain_name** is the name of the virtual machine that we want to connect to via the serial console.

Advanced troubleshooting tools

Depending on the subject – networking, hardware and software problems, or specific application problems – there are different tools that we can use to troubleshoot problems in our environ-

ments. Let's briefly go over some of these methods, with the chapters of this book in mind as we troubleshoot:

- oVirt problems
- Problems with snapshots and templates
- Virtual machine customization issues
- Ansible issues
- OpenStack problems
- Eucalyptus and AWS combo problems
- ELK stack issues

Interestingly enough, one thing that we usually don't have problems with when dealing with KVM virtualization is networking. It's really well documented – from KVM bridges all the way to open vSwitch – and it's just the matter of following the documentation. The only exception is related to firewall rules, which can be a handful, especially when dealing with oVirt and remote database connections while keeping a minimal security footprint. If you're interested in this, make sure that you check out the following link:

[https://www.ovirt.org/documentation/installing_ovirt_as_a_standalone_requirements SM remoteDB deploy](https://www.ovirt.org/documentation/installing_ovirt_as_a_standalone_requirements_SM_remoteDB_deploy).

There's a big table of ports later in that article describing which port gets used for what and which protocols they use. Also, there's a table of ports that need to be configured at the oVirt host level. We recommend that you use this article if you're putting oVirt into production.

oVirt

There are two common problems that we often encounter when dealing with oVirt:

- *Installation problems*: We need to slow down when we're typing installation options into the engine setup and configure things correctly.
- *Update problems* : These can either be related to incorrectly updating oVirt or the underlying system.

Installation problems are fairly simple to troubleshoot as they usually happen when we're just starting to deploy oVirt. This means that we can afford the luxury of just stopping the installation process and starting from scratch. Everything else will just be too messy and complicated.

Update problems, however, deserve a special mention. Let's deal with both subsets of oVirt update issues and explain them in a bit more detail.

Updating the oVirt Engine itself requires doing the thing that most of us just dislike doing – reading through heaps and heaps of documentation. The first thing that we need to check is which version of oVirt are we running. If we're – for example – running version 4.3.0 and we want to upgrade to 4.3.7, this is a minor update path that's pretty straightforward. We need to back up our oVirt database first:

```
engine-backup --mode=backup --  
file=backupfile1 --log=backup.log
```

We do this just as a precaution. Then, later on, if something does get broken, we can use the following command:

```
engine-backup --mode=restore --  
log=backup.log --file=backupfile1 --  
provision-db --provision-dwh-db --no-  
restore-permissions
```

If you didn't deploy the DWH service and its database, you can ignore the **--provision-dwh-db** option. Then, we can do the standard procedure:

```
engine-upgrade-check  
yum update ovirt-*setup*  
engine-setup
```

This should take about 10 minutes and cause no harm at all. But it's still better to be safe than sorry and back up the database before doing that.

If we're, however, migrating from some older version of oVirt to the latest one – let's say, from version 4.0.0, or 4.1.0, or 4.2.0 to version 4.3.7 – that's a completely different procedure. We need to go to the [ovirt.org](https://www.ovirt.org) website and read through the documentation. For example, let's say that we're updating from 4.0 to 4.3. There's documentation on [ovirt.org](https://www.ovirt.org) that describes all these processes. You can start here:

https://www.ovirt.org/documentation/upgrade_guide/.

This will give us 20 or so substeps that we need to complete to successfully upgrade. Please be careful and patient, as these steps are written in a very clear order and need to be implemented that way.

Now that we've covered oVirt troubleshooting in terms of upgrading, let's delve into *OS and pack-*

age upgrades as that's an entirely different discussion with much more to consider.

Keeping in mind that oVirt has its own prerequisites, ranging from CPU, memory, and storage requirements to firewall and repository requirements, we can't just blindly go and use a system-wide command such as the following:

```
yum -y update
```

We can't expect oVirt to be happy with that. It just won't, and this has happened to us many times, both in production environments and while writing this book. We need to check which packages are going to be deployed and check if they're in some co-dependent relationship to oVirt. If there are such packages, you need to make sure that you do the engine-backup procedure that we mentioned earlier in this chapter. It will save you from a lot of problems.

It's not only the oVirt Engine that can be a problem – *updating KVM hosts* that oVirt has in its inventory can also be quite a bit melodramatic. The oVirt agent (**vdsm**) that gets deployed on hosts either by the oVirt Engine or our manual installation procedures, as well as its components, also have their own co-dependencies that can be affected by a system-wide **yum -y update** command. So, put the handbrake on before just accepting upgrades as it might bring a lot of pain later. Make sure that you check the **vdsm** logs (usually located in the **/var/log/vdsm** directory). These log files are very helpful when you're trying to decipher what went wrong with **vdsm**.

oVirt and KVM storage problems

Most storage problems that we come across are usually related to either LUN or share presentation to hosts. Specifically, when you're dealing with block storage (Fibre Channel or iSCSI), we need to make sure that we don't zone out or mask out a LUN from the host, as the host won't see it. The same principle applies to NFS shares, Gluster, CEPH, or any other kind of storage that we're using.

The most common problem apart from these pre-configuration issues is related to failover – a scenario where a path toward a storage device fails. That's when we are very happy if we scaled out our storage or storage network infrastructure a bit – we added additional adapters, additional switches, configured multipathing (MPIO), and so on. Make sure that you check your storage device vendor's documentation and follow along with the best practices for a specific storage device. Believe us when we say this – iSCSI storage configuration and its default settings are a world apart from configuring Fibre Channel storage, especially when multipathing is concerned. For example, when using MPIO with iSCSI, it's much happier and snappier if you configure it properly. You'll find more details about this process in the *Further reading* section at the end of this chapter.

If you're using IP-based storage, make sure that multiple paths toward your storage device(s) use separate IP subnets as everything else is a bad idea. LACP-like technologies and iSCSI don't

work in same sentence together and you'll be troubleshooting a technology that's not meant for storage connections and is working properly, while you're thinking that it's not. We need to know what we're troubleshooting; otherwise, troubleshooting makes no sense. Creating LACP for iSCSI equals still using one path for iSCSI connections, which means wasting network connectivity that doesn't actively get used except for in the case of a failover. And you don't really need LACP or similar technologies for that. One notable exception might be blade servers as you're really limited in terms of upgrade options on blades. But even then, the solution to the *we need more bandwidth from our host to storage* problem is to get a faster network or Fibre Channel adapter.

Problems with snapshots and templates – virtual machine customization

To be quite honest, over the years of working on various virtualization technologies, which covers Citrix, Microsoft, VMware, Oracle, and Red Hat, we've seen a lot of different issues with snapshots. But it's only when you start working in enterprise IT and see how complicated operational, security, and backup procedures are that you start realizing how hazardous a *simple* procedure such as creating a snapshot can be.

We've seen things such as the following:

- The backup application doesn't want to start because the virtual machine has a snapshot (a

common one).

- A snapshot doesn't want to delete and assemble.
- Multiple snapshots don't want to delete and assemble.
- A snapshot crashes a virtual machine for quirky reasons.
- A snapshot crashes a virtual machine for valid reasons (lack of disk space on storage)
- A snapshot crashes an application running in a virtual machine as that application doesn't know how to tidy itself up before the snapshot and goes into a dirty state (VSS, sync problems)
- Snapshots get lightly misused, something happens, and we need to troubleshoot
- Snapshots get heavily misused, something always happens, and we need to troubleshoot

This last scenario occurs far more often than expected as people really tend to flex their muscles regarding the number of snapshots they have if they're given permission to. We've seen virtual machines with 20+ snapshots running on a production environment and people complaining that they're slow. All you can do in that situation is breathe in, breathe out, shrug, and ask, "What did you expect, that 20+ snapshots are going to increase the speed of your virtual machine"?

Through it all, what got us through all these issues was three basic principles:

- Really learning how snapshots work on any given technology.
- Making sure that, every time we even think of using snapshots, we first check the amount of

available storage space on the datastore where the virtual machine is located, and then check if the virtual machine already has snapshots.

- Constantly repeating the mantra: *snapshots are not backups* to all of our clients, over, and over again, and hammering them with additional articles and links explaining why they need to lay off the snapshots, even if that means denying someone permission to even take a snapshot.

Actually, this last one has become a de facto policy in many environments we've encountered.

We've even seen companies implementing a flat-out policy when dealing with snapshots, stating that the company policy is to have one or two snapshots, max, for a limited period of time. For example, in VMware environments, you can assign a virtual machine advanced property that sets the maximum number of snapshots to 1 (using a property called **snapshot.maxSnapshots**).

In KVM, you're going to have to use storage-based snapshots for these situations and hope that the storage system has policy-based capabilities to set the snapshot number to something.

However, this kind of goes against the idea of using storage-based snapshots in many environments.

Templating and virtual machine customization is another completely separate world of troubleshooting. Templating only rarely creates issues, apart from the warning we mentioned in ***Chapter 8, Creating and Modifying VM Disks, Templates, and Snapshots***, related to the serial

use of **sysprep** on Windows machines. Creating Linux templates is pretty straightforward nowadays, and people use either **virt-sysprep**, **sys-unconfig**, or custom scripts to do that. But the next step – related to virtual machine customization – is a completely different thing. This is especially true when using cloudbase-init, as cloud-init has been a standard method used for pre-configuring Linux virtual machines in cloud environments for years.

The following is a short list containing some of problems that we had with cloudbase-init:

- Cloudbase-init failed due to **Cannot load user profile: the device is not ready.**
- Domain join doesn't work reliably.
- Error during network setup.
- Resetting Windows passwords via cloudbase-init.
- Getting cloudbase-init to execute a PowerShell script from a specified directory.

The vast majority of these and other problems are related to the fact that cloudbase-init has documentation that's really bad. It does have some config file examples, but most of it is more related to APIs or the programmatic approach than actually explaining how to create some kind of configuration via examples.

Furthermore, we had various issues with different versions, as we mentioned in ***Chapter 10, Automated Windows guest deployment and customization***. We then settled on a pre-release version, which worked out-of-the-box with a configuration file that wasn't working on a stable re-

lease. But by and large, the biggest issue we had while trying to make it work was related to making it work with PowerShell properly. If we get it to execute PowerShell code properly, we can pretty much configure anything we want on a Windows-based system, so that was a big problem. Sometimes, it didn't want to execute a PowerShell script from a random directory on the Windows system disk.

Make sure that you use examples in this book for your starting points. We deliberately made examples in ***Chapter 10***, *Automated Windows guest deployment and customization* as simple as possible, which includes the executed PowerShell code. Afterward, spread your wings and fly – do whatever needs to be done with it. PowerShell makes everything easier and more natural when you're working with Microsoft-based solutions, both local and hybrid ones.

Problems working with Ansible and OpenStack

Our first interaction with Ansible and OpenStack happened years ago – Ansible was introduced in 2012, and OpenStack in 2010. We always thought that both were (are) very cool pieces of kit, albeit with a few problems. Some of these small niggles were related to the fast pace of development (OpenStack), with a large number of bugs being solved from version to version.

In terms of Ansible, we had loads of fights with people over it – one day, the subject was related to the fact that *we're used to using Puppet, why do*

we need Ansible?!; the next day it was argh, this syntax is so complex; the day after that it was something else, and something else... and it was usually just related to the fact that the Ansible architecture is much simpler than all of them in terms of architecture, and a bit more involved – at least initially – in terms of syntax. With Ansible, it's all about the syntax, as we're sure that you either know or will find out soon enough.

Troubleshooting Ansible playbooks is usually a process that has a 95% chance that we misspelled or mistyped something in the configuration file. We're talking about the initial phase in which you already had a chance to work with Ansible for a while. Make sure that you re-check outputs from Ansible commands and use their output for that. In that sense, it's really excellent. You don't need to do complex configuration (such as with **libvirt**, for example) to get usable output from your executed procedures and playbooks. And that makes our job a lot easier.

Troubleshooting OpenStack is a completely different can of worms. There are some well-documented OpenStack problems out there, which can also be related to a specific device. Let's use one example of that – check out the following link for issues when using NetApp storage:

https://netapp-openstack-dev.github.io/openstack-docs/stein/appendices/section_common-problems.html.

The following are some examples:

- Creating volume fails
- Cloning volume fails
- Volume attachment fails
- Volume upload to image operation fails
- Volume backup and/or restore fails

Then, for example, check out these links:

- <https://docs.openstack.org/cinder/queens/configuration/block-storage/drivers/ibm-storwize-svc-driver.html>
- https://www.ibm.com/support/knowledgecenter/STHGUJ_8.2.1/com.ibm

As you've probably deduced yourself, OpenStack is really, really picky when it comes to storage. That's why storage companies usually create reference architectures for their own storage devices to be used in OpenStack-based environments. Check out these two documents from HPE and Dell EMC as good examples of that approach:

- <https://www.redhat.com/cms/managed-files/cl-openstack-hpe-synergy-ceph-reference-architecture-f18012bf-201906-en.pdf>
- <https://docs.openstack.org/cinder/rocky/configuration/block-storage/drivers/dell-emc-unity-driver.html>

One last word of warning relates to the most difficult obstacle to surmount – OpenStack version upgrades. We can tell you loads of horror stories on this subject. That being said, we're also partially to blame here, because we, as users, deploy various third-party modules and utilities (vendor-based plugins, forks, untested solutions, and so on), forget about using them, and then we're really surprised and horrified when the upgrade procedure fails. This goes back to our multiple

discussions about documenting environments that we had throughout this book. This is a subject that we'll revisit for one final time just a bit later in this chapter.

Dependencies

Every administrator is completely aware that almost every service has some dependencies – either the services that depend on this particular service running or services that our service needs to work. Dependencies are also a big thing when working with packages – the whole point of package managers is to strictly pay attention to what needs to be installed and what depends on it so that our system works as it should.

What most admins do wrong is forget that, in larger systems, dependencies can stretch across multiple systems, clusters, and even data centers.

Every single course that covers OpenStack has a dedicated lesson on starting, stopping, and verifying different OpenStack services. The reason for this is simple – OpenStack is usually run across a big number of nodes (hundreds, sometimes thousands). Some services must run on every node, some are needed by a set of nodes, some services are duplicated on every node instance, and some services can only exist as a single instance.

Understanding the basics of each service and how it falls into the whole OpenStack schema is not only essential when installing the whole system but is also the most important thing to know

when debugging why something is not working on OpenStack. Read the documentation at least once to *connect the dots*. Again, the *Further reading* section at the end of this chapter contains links that will point you in the right direction regarding OpenStack.

OpenStack is one of those systems that includes *how do I properly reboot a machine running X?* in the documentation. The reason for this is as simple as the whole system is complex – each part of the system both has something it depends on and something that is depending on it – if something breaks, you need to not only understand how this particular part of the system works, but also how it affects everything else. But there is a silver lining through all this – in a properly configured system, a lot of it is redundant, so sometimes, the easiest way of repairing something is to reinstall it.

And this probably sums the whole troubleshooting story – trying to fix a simple system is usually more complicated and time-consuming than fixing a complex system. Understanding how each of them works is the most important part.


Troubleshooting Eucalyptus

It would be a lie to say that once we started the installation process, everything went according to the manual – most of it did, and we are reasonably sure that if you follow the steps we documented, you will end up with a working service or system, but at any point in time, there are things that can – and will – go wrong. This is

when you need to do the most complicated thing imaginable – troubleshoot. But how do you do that? Believe it or not, there is a more or less systematic approach that will enable you to troubleshoot almost any problem, not just KVM/OpenStack/AWS/Eucalyptus-related ones.

Gathering information

Before we can do anything, we need to do some research. And this is the moment most people do the wrong thing, because the obvious answer is to go to the internet and search for the problem. Take a look at this screenshot:



```

- execute the ruby block Upload cluster keys Chef Server
* execute[Register User Facing 192.168.5.48] action run[2020-04-09T16:38:58-04:00] INFO: Processing
g execute[Register User Facing 192.168.5.48] action run (eucalyptus::register-components line 99)
[2020-04-09T16:38:58-04:00] INFO: Processing execute[Guard resource] action run (dynamically defined
)
[2020-04-09T16:38:58-04:00] INFO: execute[Register User Facing 192.168.5.48] ran successfully

- execute eval 'cloudadmin-assume-system-credentials' && //usr/bin/euserv-register-service -t use
r-api -h 192.168.5.48 API_192.168.5.48
* execute[Register Walrus] action run[2020-04-09T16:38:58-04:00] INFO: Processing execute[Register
Walrus] action run (eucalyptus::register-components line 110)
[2020-04-09T16:38:58-04:00] INFO: Processing execute[Guard resource] action run (dynamically defined
)
[2020-04-09T16:38:58-04:00] INFO: execute[Register Walrus] ran successfully

- execute eval 'cloudadmin-assume-system-credentials' && //usr/bin/euserv-register-service -t wal
rusbackend -h 192.168.5.48 walrus-0
Recipe: eucalyptus::walrus
* yum_package[eucalyptus-walrus] action upgrade[2020-04-09T16:38:58-04:00] INFO: Processing yum_pa
ckage[eucalyptus-walrus] action upgrade (eucalyptus::walrus line 23)
[2020-04-09T16:39:04-04:00] INFO: yum_package[eucalyptus-walrus] installing eucalyptus-walrus-4.4.5-
0.34.as.el7 from eucalyptus repository
[2020-04-09T16:39:07-04:00] INFO: yum_package[eucalyptus-walrus] upgraded eucalyptus-walrus to 4.4.5-
0.34.as.el7

- upgrade package eucalyptus-walrus from uninstalled to 4.4.5-0.34.as.el7
[2020-04-09T16:39:07-04:00] INFO: yum_package[eucalyptus-walrus] sending create action to templatefe
ucalyptus.conf] (immediate)
Recipe: eucalyptus::storage-controller
* template[eucalyptus.conf] action create[2020-04-09T16:39:07-04:00] INFO: Processing templatefe
ucalyptus.conf] action create (eucalyptus::storage-controller line 58)
[2020-04-09T16:39:07-04:00] INFO: template[eucalyptus.conf] backed up to /root/.chef/local-mode-cach
e/backup/etc/eucalyptus/eucalyptus.conf.chef-28200409163907.862583
[2020-04-09T16:39:07-04:00] INFO: template[eucalyptus.conf] updated file contents //etc/eucalyptus/e
ucalyptus.conf

```

Figure 16.5 – Eucalyptus logs, part I – clean, crisp, and easy to read – every procedure that's been done in Eucalyptus clearly visible in the log

If you haven't noticed already, the internet is full of ready-made solutions to almost any imaginable problem, with a lot of them being wrong. There are two reasons why this is so: most of the people who worked on the solution didn't understand what the problem was, so as soon as they found any solution that solved their particular problem, they simply stopped solving it. In other

words – a lot of people in IT try to picture a path from point A (problem) to point B (solution) as a laser beam – super flat, the shortest possible path, no obstacles along the way. Everything is nice and crisp and designed to mess with our troubleshooting thought process as soon as the laser beam principle stops working. This is because, in IT, things are rarely that simple.

Take, for example, any problem caused by DNS being misconfigured. Most of those can be *solved* by creating an entry in the *hosts* file. This solution usually works but is, at the same time, wrong on almost any level imaginable. The problem that is solved by this is solved on only one machine – the one that has the particular hosts file on it. And the DNS is still misconfigured; we just created a quick, undocumented workaround that will work in our particular case. Every other machine that has the same problem will need to be patched in this way, and there is a real possibility that our fix is going to create even more problems down the road.

The real solution would obviously be to get to the root of the problem itself and solve the issue with DNS, but solutions like this are few and far between on the internet. This happens mainly because the majority of the commenters on the internet are not familiar with a lot of services, and quick fixes are basically the only ones they are able to apply.

Another reason why the internet is mostly wrong is because of the famous *reinstall fixed the problem* solution. Linux has a better track record

there as people who use it are less inclined to solve everything by wiping and reinstalling the system, but most of the solutions you will find for Windows problems are going to have at least one simple *reinstall fixed it*. Compared to just giving a random fix as the one that always works, this *reinstall* approach is far worse. Not only does it mean you are going to waste a lot of time reinstalling everything; it also means your problem may or may not be solved in the end, depending on what the problem actually was.

So, the first short piece of advice we will give is, *do not blindly trust the internet*.

OK, but what should you actually do? Let's take a look:

1. *Gather information about the problem.* Read the error message, read the logs (if the application has logs), and try to turn on debug mode if at all possible. Get some solid data. Find out what is crashing, how it is crashing, and what problems are causing it to crash. Take a look at the following screenshot:

```
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/tutorials/launch-instances.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/bind-addr.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/node-template.json in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/muke.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/eucalyptus_helper.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/enterprise.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/midonet.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/create_riakcs_user.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/motherbrain.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/tutorials/install-image.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/cloud-controller.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/libraries/ceph.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/cloud-service.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/install-nc.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/metadata.json in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/faststart/cloud-in-a-box.sh in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/create-first-resource-s.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/cluster-controller.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/eucanetd.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/eucanetd.rb in the cache.
[2020-04-09T16:34:24-04:00] INFO: Storing updated cookbooks/eucalyptus/recipes/eucanetd.rb in the cache.
```

Figure 16.6 – Eucalyptus logs, part II – again, clean, crisp, and easy to read – information messages about what was updated and where

2. *Read the documentation.* Is the thing you are trying to do even supported? What are the prerequisites for the functioning system? Are you missing something? A cache disk? Some amount of memory? A fundamental service that is a dependency for your particular system? A dependency that's a library or additional packages? A firmware upgrade?

Sometimes, you will run into an even bigger problem, especially in poorly written documentation – *some crucial system dependency may be mentioned in passing* and may cause your entire system to crash. Take, for example, an external identification service – maybe your directory uses a *wrong character set*, causing your system to crash when a particular user uses it in a particular way. Always make sure you understand how your systems are interconnected.

Next, check your system. If you are installing a new system, check the prerequisites. Do you have *enough disk space* and *memory*? Are all the services your application requires readily available and working properly?

Search the internet. We mentioned previously that the internet has a simple, incorrect solution to all possible problems, but it usually also has the right solution hidden somewhere among the wrong ones. Having armed yourself with a lot of data about your particular system and your specific problem, the internet will soon become

your friend. Since you understand what the problem is, you will be able to understand what solutions have been offered to you are simply wrong.

Now, let's talk about a real-world problem we created while installing Eucalyptus on purpose, just to show you how important documentation is.

We showed you how to install Eucalyptus in ***Chapter 13***, *Scaling Out KVM with AWS* – we not only went through the installation process but also how to use this amazing service. If you want to learn something about how not to do it, continue reading. We will present you with a deliberate scenario of an unsuccessful Eucalyptus installation that won't finish because we *creatively forgot to do some steps that we knew we needed to do*. Let's put it this way – we acted as humans and used the method of *browsing the documentation* instead of *actually sitting down and reading the documentation*. Does that sound familiar?

Installing Eucalyptus should be a straightforward task since its installation is, in essence, an exercise in applied scripting. Eucalyptus even says so on the front page of the project: *just run this script*.

But the truth is much more complicated – Eucalyptus can definitely be installed using only this script, but certain prerequisites must be met. Of course, in your rush to test the new service, you will probably neglect to read the documenta-

tion, as we did, since we already had experience with Eucalyptus.

We configured the system, we started the installation, and we ran into a problem. After confirming the initial configuration steps, our installation failed with an error that said it was unable to resolve a particular address:

192.168.1.1.nip.io.

DNS is one of the primary sources of problems in the IT infrastructure, and we quickly started debugging – the first thing we wanted to see was what this particular address is. There's actually a saying in IT – *It's always DNS*. It looks like a local address, so we started pinging it, and it seemed fine. But why is DNS even involved with IP addresses? DNS should be resolving domain names, not IP addresses. Then, we turned to the documentation, but that didn't yield much. The only thing that we found was that DNS must work for the whole system to work.

Then, it was time to try and debug the DNS. First, we tried resolving it from the machine we were installing it on. The DNS returned a timeout. We tried this on another machine and we got back the response we didn't expect –

127.0.0.1.nip.io resolved as **127.0.0.1**, which meant localhost. Basically, we asked a DNS on the internet to give us an address, and it directed us to our local system.

So, we had an error we didn't understand, an address that resolved to an IP address we hadn't expected, and two different systems exhibiting

completely different behaviors for an identical command. We turned our attention to the machine we were installing on and realized it was misconfigured – there was no DNS configured at all. The machine not only failed to resolve our *strange* IP address but failed to resolve anything.

We fixed that by pointing to the right DNS server. Then, in true IT fashion, we restarted the installation so that we were able to go through with this part and everything was ok, or so it seemed. But what happened? Why is a local service resolving such strange names and why do they get resolved at all?

We turned to the internet and took a look at the name of the domain that our mystery name had at its end. What we found out is that the service, **nip.io**, actually does just the thing we observed it do – when asked for a particular name formed from an IP address in the local subnet range (as defined by **RFC 1918**), it returned that same IP.

Our next question was – why?

After some more reading, you will realize what the trick was here – Eucalyptus uses DNS names to talk to all of its components. The authors very wisely chose not to hardcode a single address into the application, so all the services and nodes of the system have to have a real DNS registered name. In a normal multi-node, multi-server installation, this works like a charm – every server and every node are first registered with their appropriate DNS server, and Eucalyptus will try

and resolve them so it can communicate with the machine.

We are installing a single machine that has all the services on it, and that makes installing easier, but nodes do not have separate names, and even our machine may not be registered with the DNS. So, the installer does a little trick. It turns local IP addresses into completely valid domain names and makes sure we can resolve them.

So, now we know what happened (resolving process was not working) and why it happened (our DNS server settings were broken), but we also understood why DNS was needed in the first place.

This brings us to the next point – *do not presume anything*.

While we were troubleshooting and then following up on our DNS problem, our installation crashed. Eucalyptus is a complex system and its installation is a fairly complex thing – it automatically updates the machine you run it on, then it installs what seems like thousands of packages, and then it downloads, configures, and runs a small army of images and virtual packages. To keep things tidy, the user doesn't see everything that is happening, only the most important bits. The installer even has a nice ASCII graphic screen to keep you busy. Everything was OK up to a point, but suddenly, our installation completely crashed. All we got was a huge stack trace that looked like it belonged to the Python lan-

This is the installation screen. We can't see any real information regarding what is happening, but the third line from the top contains the most important clue – the location of the log file. In order to stop your screen from being flooded with information, the installer shows this very nice figlet-coffee graphic (everyone who ever used IRC in the 1990s and 2000s will probably smile now), but also dumps everything that is happening into a log. By everything, we mean everything – every command, every input, and every output. This makes debugging easy – we just need to scroll to the end of this file and try to go from that point backward to see what broke. Once we did that, the solution was simple – we forgot to allocate enough memory for the machine. We gave it 8 GB of RAM, and officially it should have at least 16 GB to be able smoothly. There are reports of machines running with as little as 8 GB of RAM, but that makes absolutely no sense – we are running a virtualized environment after all.

AWS and its verbosity, which doesn't help

Another thing we wanted to mention is AWS and how to troubleshoot it. AWS is an amazing service, but it has one huge problem – its size. There are so many services, components, and service parts that you need to use to get something to run on AWS that simple tasks can get very complicated. Our scenario involved trying to put up an EC2 instance that we used as our example.

This task is relatively straightforward and demonstrates how a simple problem can have a simple solution that can, at the same time, be completely not obvious.

Let's go back to what we were trying to do. We had a machine that was on a local disk. We had to transfer it to the cloud and then create a running VM out of it. This is probably one of the simplest things to do.

For that, we created an S3 bucket and got our machine from the local machine into the cloud. But after we tried to run the machine, all we got was an error.

The biggest problem with a service like AWS is that it is enormous and that there is no way of understanding everything at once – you must build your knowledge block by block. So, we went back to the documentation. There are two kinds of documentation on AWS – extensive help that covers every command and every option on every service, and guided examples. Help is amazing, it really is, but if you have no idea what you are looking for, it will get you nowhere. Help in this form only works as long as you have a basic understanding of the concepts. If you are doing something for the first time, or you have a problem you haven't seen before, we suggest that you find an example of the task you are trying to do, and do the exercise.

In our case, this was strange, since all we had to do was run a simple command. But our import was still failing. After a couple of hours of us

banging our head against the wall, we decided to just behave like we knew nothing and went and did the *how do I import a VM into AWS?* example. Everything worked. Then, we tried importing our own machine; that didn't work. The commands were copy/pasted, but it still didn't work.

And then we realized the most important thing – *we need to pay attention to details*. Without this train of thought properly implemented and executed, we're inviting a world of problems upon ourselves.

Paying attention to details

To cut the story (that's way too long) short, what we did wrong was we misconfigured the identity service. In a cloud environment such as AWS, every service runs as an independent domain, completely separate from other services. When something needs to be done, the service doing it has to have some kind of authorization. There is a service that takes care of that – IAM – and the obvious default for every request from every service is to deny everything. Once we decide what needs to be done, it is our job to configure proper access and authorization. We knew that, and so we created all the roles and permissions for EC2 to access the files in S3. Even though that may sound strange, we had to give a service we are using the permission to get the files we uploaded. If you are new to, this you might expect this to be automatic, but it isn't.

Check out the following small excerpt, which is from the really long list of roles that AWS has

predefined. Keep in mind that the complete list is much, much longer and that we've barely scratched the surface of all of the available roles. These are just roles that have names starting with the letter A:

Create policy

Policy actions

Filter policies

Search




















	Policy name	Type
<input type="radio"/>	 AccessAnalyzerServiceRolePolicy	AWS managed
<input type="radio"/>	 AdministratorAccess	Job function
<input type="radio"/>	 AlexaForBusinessDeviceSetup	AWS managed
<input type="radio"/>	 AlexaForBusinessFullAccess	AWS managed
<input type="radio"/>	 AlexaForBusinessGatewayExecution	AWS managed
<input type="radio"/>	 AlexaForBusinessNetworkProfileServicePolicy	AWS managed
<input type="radio"/>	 AlexaForBusinessPolyDelegatedAccessPolicy	AWS managed
<input type="radio"/>	 AlexaForBusinessReadOnlyAccess	AWS managed
<input type="radio"/>	 AmazonAPIGatewayAdministrator	AWS managed
<input type="radio"/>	 AmazonAPIGatewayInvokeFullAccess	AWS managed
<input type="radio"/>	 AmazonAPIGatewayPushToCloudWatchLogs	AWS managed
<input type="radio"/>	 AmazonAppStreamFullAccess	AWS managed
<input type="radio"/>	 AmazonAppStreamReadOnlyAccess	AWS managed
<input type="radio"/>	 AmazonAppStreamServiceAccess	AWS managed
<input type="radio"/>	 AmazonAthenaFullAccess	AWS managed
<input type="radio"/>	 AmazonAugmentedAIFullAccess	AWS managed
<input type="radio"/>	 AmazonAugmentedAIHumanLoopFullAccess	AWS managed
<input type="radio"/>	 AmazonAugmentedAIIntegratedAPIAccess	AWS managed
<input type="radio"/>	 AmazonChimeFullAccess	AWS managed

Figure 16.8 – AWS predefined roles

What we misconfigured was the name of the role – to import the VM into the EC2 instance, there needs to be a security role named **vmimport** giving EC2 the right permissions. We configured a role named **importvm** in our haste. When we completed the examples, we pasted the examples and everything was fine, but as soon as we started using our security settings, EC2 was failing to do its job. So, always *check the product documentation and read it carefully*.

Troubleshooting problems with the ELK stack

The ELK stack can be used to monitor our environment efficiently. It does require a bit of manual labor, additional configuration, and being a bit sneaky, but it can still offer reporting, automatic reporting, sending reports via email, and a whole lot of other valuable things.

Out of the box, you can't just send reports directly – you need to do some more snooping. You can use Watcher, but most of the functionality you need from it is commercial, so you'll have to spend some cash on it. There are some other methods, as well:

- Using snapshot for Kibana/Grafana – check out this URL: <https://github.com/parvez/snapshot>
- Using ElastAlert – check out this URL: <https://github.com/Yelp/elastalert>
- Use Elastic Stack Features (formerly X-Pack) – check out this URL: <https://www.elastic.co/guide/en/x-pack/current/installing-xpack.html>

Here's one more piece of advice: you can always centralize logs via **rsyslog** as it's a built-in feature. There are free applications out there for browsing through log files if you create a centralized log server (Adiscon LogAnalyzer, for example). If dealing with ELK seems like a bit too much to handle, but you're aware of the fact that you need something, start with something like that. It's very easy to install and configure and offers a free web-like interface with regular ex-

pression support so that you can browse through log entries.

Best practices for troubleshooting KVM issues

There are some common-sense best practices when approaching troubleshooting KVM issues. Let's list some of them:

- *Keep it simple, in configuration:* What good does a situation in which you deployed 50 OpenStack hosts across three subnets in one site do? Just because you can subnet to an inch of an IP range's life doesn't mean you should. Just because you have eight available connections on your server doesn't mean that you should LACP all of them to access iSCSI storage. Think about end-to-end configuration (for example, Jumbo Frames configuration for iSCSI networks). Simple configuration almost always means simpler troubleshooting.
- *Keep it simple, in troubleshooting:* Don't go chasing the super-complex scenarios first. Start simple. Start with log files. Check what's written there. With time, use your gut feeling as it will develop and you'll be able to trust it.
- *Use monitoring tools such as ELK stack:* Use something to monitor your environments constantly. Invest in some kind of large-screen display, hook it up to a separate computer, hang that display on a wall, and spend time configuring important dashboards for your environments.

- *Use reporting tools* to create multiple auto-generated reports about the state of your environment: Kibana supports report generation, for example, in PDF format. As you monitor your environment, you will notice some of the *more sensitive* parts of your environments, such as storage. Monitor the amount of available space. Monitor path activity and network connections being dropped from host to storage. Create reports and send them automatically to your email. There's a whole world of options there, so use them.
- *Create notes while you configure your environment*: If nothing else, do this so that you have some starting point and/or a reference for future, as there will be many changes that are often done *on the fly*. And when the process of taking notes is finished, create documentation.
- *Create documentation*: Make these permanent, readable, and as simple as possible. Don't *remember* things, *write things down*. Make it a mission to write everything down, and try to spread that culture all around you.

Get used to having a large portion of **<insert your favorite drink here>** available at all times and many sleepless nights if you want to work in IT as administrator, engineer, or DevOps engineer. Coffee, Pepsi, Coca-Cola, lemon juice, orange juice.... whatever gets your intellectual mojo flowing. And sometimes, learn to walk away from a problem for a short period of time. Solutions often click in your head when you're thinking about something completely opposite to work.

And finally, remember to try and have fun while working. Otherwise, the whole ordeal of working with KVM or any other IT solution is just going to be an *Open Shortest Path First* to relentless frustration. And frustration is never fun. We prefer yelling at our computers or servers. It's therapeutic.

Summary

In this chapter, we tried to describe some basic troubleshooting steps that can be applied generally and when troubleshooting KVM. We also discussed some of the problems that we had to deal with while working with various subjects of this book – Eucalyptus, OpenStack, the ELK stack, cloudbase-init, storage, and more. Most of these issues were caused by misconfiguration, but there were quite a few where documentation was severely lacking. Whatever happens, don't give up. Troubleshoot, make it work, and celebrate when you do.

Questions

1. What do we need to check before deploying the KVM stack?
2. What do we need to configure after deploying the KVM stack in terms of making sure that virtual machines are going to run after reboot?
3. How do we check KVM guest log files?
4. How can we turn on and configure KVM debug logging permanently?
5. How can we turn on and configure KVM debug logging at runtime?

6. What's the best way to solve oVirt's installation problems?
7. What's the best way to solve oVirt's minor and major version upgrade problems?
8. What's the best way to manage the oVirt Engine and host updates?
9. Why do we need to be careful with snapshots?
10. What are the common problems with templates and cloudbase-init?
11. What should be our first step when installing Eucalyptus?
12. What kind of advanced capabilities for monitoring and reporting can we use with the ELK stack?
13. What are some of the best practices when troubleshooting KVM-based environments?

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- Working with KVM debug logging:
<https://wiki.libvirt.org/page/DebugLogs>
- Firewall requirements for oVirt nodes and oVirt Engine:
https://www.ovirt.org/documentation/installing_ovirt_as_a_standalone_requirements_SM_remoteDB_deploy
- oVirt upgrade guide:
https://www.ovirt.org/documentation/upgrade_guide/
- Common problems with NetApp and Openstack integration: <https://netapp-openstack-dev.github.io/openstack->

[docs/stein/appendices/section_common-problems.html](#)

- Integrating the IBM Storwize family and SVC driver in OpenStack:
[https://docs.openstack.org/cinder/queens/configuration/block-storage/drivers/ibm-storwize-svc-driver.html](#)
- Integrating IBM Storwize and OpenStack:
[https://www.ibm.com/support/knowledgecenter/STHGuj_8.2.1/com.ibm](#)
- HPE Reference Architecture for the Red Hat OpenStack Platform on _HPE Synergy with Ceph Storage:
[https://www.redhat.com/cms/managed-files/cl-openstack-hpe-synergy-ceph-reference-architecture-f18012bf-201906-en.pdf](#)
- Integrating Dell EMC Unity and OpenStack:
[https://docs.openstack.org/cinder/rocky/configuration/block-storage/drivers/dell-emc-unity-driver.html](#)
- DM-multipath configuration for Red Hat Enterprise Linux 7:
[https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/dm_multipath/mpio_setup](#)
- DM-multipath configuration for Red Hat Enterprise Linux 8:
[https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/pdf/configuring_device_mapper_multipath-en-8-Configuring_device_mapper_multipath-en-US.pdf](#)
- Using snapshot for Kibana/Grafana:
[https://github.com/parvez/snapshot](#)
- Using ElastAlert:
[https://github.com/Yelp/elastalert](#)
- Using Elastic Stack Features (formerly X-Pack):
[https://www.elastic.co/guide/en/elasticsearch/reference/current/setup_xpack.html](#)

- Troubleshooting OpenStack Networking:
<https://docs.openstack.org/operations-guide/ops-network-troubleshooting.html>
- Troubleshooting OpenStack Compute:
<https://docs.openstack.org/ocata/admin-guide/support-compute.html>
- Troubleshooting OpenStack Object Storage:
<https://docs.openstack.org/ocata/admin-guide/objectstorage-troubleshoot.html>
- Troubleshooting OpenStack Block Storage:
<https://docs.openstack.org/ocata/admin-guide/blockstorage-troubleshoot.html>
- Troubleshooting OpenStack Shared File Systems:
<https://docs.openstack.org/ocata/admin-guide/shared-file-systems-troubleshoot.html>
- Troubleshooting a Bare Metal OpenStack service:
<https://docs.openstack.org/ocata/admin-guide/baremetal.html#troubleshooting>