Linux Academy
Study Guide

Nagios Professional Certification

Contents

Introduction

About This Course

In this course, we will learn about and practice using Nagios Core 4.2.x, which is one of the most widely used monitoring packages in the industry. Underneath its apparent simplicity, Nagios Core provides powerful tools to monitor devices and services, perform checks, handle events, generate notifications, and integrate with existing infrastructure and operating procedures. Fortunately, the fundamental concepts of Nagios are quite simple, as we'll see shortly.

What You Need for This Course

- **This study guide** • This guide is available as a PDF in the **Downloads** tab. Save it and read it!
- **Lab servers** • We will use several lab servers for this course, all of which are based the Nagios CentOS 7 Lab Server.

What is Provided

In addition to this study guide, the lab servers for this course have the Nagios Core Software and the official Nagios plugins installed.

Additional Resources and Tools

The Linux Academy Community is a great place to get guidance and assistance from other students, as well as Linux Academy instructors – you are also quite welcome to share your knowledge with others, too. We really enjoy hearing from our students, so feel free to make use of the community at any time.

What You Will Learn in This Course

This course will take you through the configuration and use of Nagios so that you will be able to effectively use Nagios from both the shell and the web GUI. This includes:

- How to author the various objects used in Nagios
- Understanding checks, events, and notifications
- A solid understanding of Nagios' various components
- Why Nagios is so flexible
- The necessary knowledge and skills to sit for the Nagios Certified Professional exam

What is Nagios?

Nagios is a suite of binaries and utilities used to monitor devices on your network (servers, switches, routers, etc.), and the services they provide (SSH, WWW, etc.). Nagios is able to monitor these both actively (by using check processes that originate from the Core Host, where the Nagios Core software is installed) and passively (by accepting information that originates from devices and services).

Nagios Core utilizes compiled and binaries and scripts, the utility of which is facilitated by an ingenious abstraction layer, to perform checks. Because of its extensibility and reliance on open software and standards, Nagios Core is fairly easy to integrate into existing infrastructure and operating procedures.

Nagios can also gather performance information, if so configured. All of the information gathered by Nagios can be stored in a relational database for historical and analytical purposes, although we do not endeavour such in this course.

Open Source vs. Commerical Editions

Nagios Core is the open-source verson of Nagios. The source code is available to the community. While Nagios XI can be used on a limited basis for free (up to seven nodes at the time of writing), because the Nagios Certified Professional Exam covers Nagios Core, that is what we use in this course.

Nagios Core forms the foundation of Nagios XI, and Nagios is committed to maintaining Nagios Core.

All versions of Nagios may be administered from the command line.

Support

Support contracts are available as part of commercial Nagios XI license. They are not available with Nagios Core.

Why Monitor?

Monitoring can provide early indicators of trouble before problems become critical and services are disrupted. System administrators can use tools like Nagios Core to better undertake capacity planning, to handle situations proactively (rather than reactively), and maintain a better "big picture" of system and network-wide operations.

Of course, there are security implications as well. Monitoring tools like Nagios can provide information that can inform an organization's ability to defend its infrastructure from and attack, or short-circuit an attack altogether. Unfortunately, when improperly configured or poorly implemented, any monitoring system also presents additional risk to security and stability; Nagios Core is no exception to that rule.

Nagios Core Strengths

Nagios Core's features can be summed up as follows:

- **Flexible** • Nagios Core provides tools to monitor just about anything you can attach to a network, as well as services and attributes which may not come to mind when one thinks of "monitoring" in the Enterprise sense. (We'll see some good examples in this course.)
- **Extensible** • Nagios Plugins rely on a uniform convention, which means it's fairly simply to write plugins that will interoperate with Nagios.
- **Scalable** • When properly managed, Nagios can grow with an organization both in terms of size and the types of devices that may be monitored.
- **Open Source** • Nagios Core is licensed under the GNU GPL V2, which means there is a vibrant community of contributors and users maintaining both the core software and the plugins.
- **Customizable** • Nagios facilitates customization through plugins which allow manipulation of almost any information reported to it.

Nagios Documentation

Nagios maintains an extensive documentation set available at: https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/toc.html

Support Forums

The Nagios Support Forums are a great place to share with others, discuss problems, and find resolutions to problems that crop up using Nagios. They do require registration to post. The forums can be found at: https://support.nagios.com/forum/

Some Definitions and Terms

Nagios' flexibility has its roots in meticulous engineering, which will be become clear as we delve into how Nagios works. We need to understand a few key terms before we continue.

Nagios Core Host

This term refers to the host on which the Nagios Core software is installed. This is also where the results of checks are sent when checks are executed by plugins. This is the first host we will configure in this course.

Plugins

Plugins are utilities external to Nagios Core (typically binaries or scripts), which are used to collect information about devices and services. This information is then reported to the Nagios Core Host. Some plugins, such as the Nagios Remote Plugin Executor (NRSE), exist solely to provide for the execution of plugins on remote systems.

Objects

Nagios employs a handful of object types; these interact to provide Nagios' functionality:

- **Commands** • Commands provide a consistent way to interact with plugins, and also encompass event handling, notifications, and checks.
- **Contacts and Contact Groups** • These objects are used to define individuals (and their contact information) who are notified in the context of specific events.
- **Hosts and Host Groups** • These define specific devices or services on the network.
- **Notifications** • These define what Contacts and Contact Groups should be notified of an event. These can be very specific; for instance, the "sysadmins" group can be notified that hosts in the "web-servers" host group are down after a specific period of time IF the outage occurs during the "business-hours."
- **Services** • These are specific services provided by devices on the network. SSH, FTP, and HTTP/80 are examples, but services can also include the attributes or properties of the host on which they run. The time of the hosts on-board clock, for instance, is considered a service. DNS records, host entries, and RAID controller cards are also examples of services.

Users

For our purposes, "system" users and groups refer to those managed in `/etc/passwd`. "Nagios" users refer to accounts created to provide access to the Nagios Web Interface and are not system accounts.

Acknowledgments

When Nagios encounters a problem, it sends notifications until the problem is resolved or some action is taken. One way to temporarily disable notifications (typically while a problem is being resolved), is for a user to acknowledge the problem.

Downtime

When it is anticipated that a host or service will be down, Nagios can be informed in advance by scheduling downtime. This keeps Nagios from generating alerts.

Disable

This refers to turning off a feature in Nagios, be it a check, a notification, or an event.

Latency

The measurement of the time between the scheduling of a check and when the check is actually run.

State

Nagios has four basic states and two *types* of states. The four basic states are:

0: OK (green)
1: WARNING (yellow)
2: CRITICAL (red)
3: UNKNOWN (orange)

Additionally, states may be "SOFT" or "HARD." This seems confusing at first, but let's talk about why the two types of state exist.

Soft states result when a host is determined to be "not OK" but Nagios Core hasn't yet rechecked enough times to be certain. (The number of times it will re-check is specified with the "max check attempts" directive in the service or host definition file.)

When a host's state changes, and the change is to a SOFT state, event handlers are executed to handle the soft state. That's all that actually happens, apart from logging. Event handlers, which are actually commands, can be used to proactively fix the problem before Nagios re-categorizes the issue as a HARD state issue.

Hard states are those which:

- Were previously soft, but have been verified through re-checks
- Were in a previously hard state (like WARNING) and have become critical
- When a service check returns a non-OK result and the host is DOWN or UNREACHABLE
- When a service or host recovers from a hard error state
- When a passive host check is received, as acceptance of soft-state passive check results are disabled by default
- There may be other cases as well, but these are fairly readily identifiable.

Additional States

In addition to the basic states, hosts and services may be indicated in one of the follow statuses:

- **r** • RECOVERED
- **f** • FLAPPING
- **s** • SCHEDULED MAINTENANCE (downtime)

Triggered Downtime

This term refers to a mechanism by which Nagios Core can place hosts and services into downtime based on ANOTHER host or service being in downtime.

Getting Started

First Log In

Nagios Core is already installed on the Nagios CentOS 7 Lab Server. Create an instance of this in the Server Labs window. Log in as *user*, and change the *user* and *root* passwords.

This first host we spin up in Server Labs is central to our efforts throughout this course, which is why we called it by a special name: this server is our "Nagios Core host."

We also need to make some adjustments to our Nagios Core host, when using the Server Lab.

Disable SELinux

Nagios has some known issues with SELinux, and digging into those remedies is outside the scope of our course. For now, we're going to disable SELinux by editing the `/etc/sysconfig/selinux` file and changing the `SELINUX=enforcing` line to the following:

```
SELINUX=disabled
```

Disabling Transparent Huge Pages

Transparent Huge Pages are a great idea, but there are some known issues with them. We are going to disable them by adding these two lines to `/etc/rc.local`:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

Additionally, make sure that `/etc/rc.local` is executable with the following command:

```
chmod +x /etc/rc.local
```

Disabling File Access Times

Adding this option to `/etc/fstab` prevents the kernel from updating the file metadata each time a file is

accessed; this should provide for faster disk I/O. We change the root mount line to:

```
UUID=####### / xfs defaults,noatime 1 1
```

Disabling Kernel Updates

Make sure that this line in `/etc/yum.conf` is *not* commented out:

```
#exclude=kernel* redhat-release*
```

Your `/etc/yum.conf` file should look like this:

```
[main]
cachedir=/var/cache/yum/$basearch/$releasever
keepcache=0
debuglevel=2
logfile=/var/log/yum.log
exactarch=1
obsoletes=1
gpgcheck=1
plugins=1
installonly_limit=5
bugtracker_url=http://bugs.centos.org/set_project.php?project_
id=23&ref=http://bugs.centos.org/bug_report_page.php?category=yum
distroverpkg=centos-release

exclude=kernel* redhat-release*
#  This is the default, if you make this bigger yum won't see if the
metadata
# is newer on the remote and so you'll "gain" the bandwidth of not
having to
# download the new metadata and "pay" for it by yum not having correct
# information.
#  It is esp. important, to have correct metadata, for distributions
like
# Fedora which don't keep old packages around. If you don't like this
checking
# interupting your command line usage, it's much better to have
something
# manually check the metadata once an hour (yum-updatesd will do this).
# metadata_expire=90m

# PUT YOUR REPOS HERE OR IN separate files named file.repo
# in /etc/yum.repos.d
```

Again, be certain the `exclude` line is *not* commented out. Depending on the cloud provider used to host your Nagios Core server, a kernel update (even a minor one) could result in your server being broken beyond recovery.

Configuring Nagios

Configuration Files

You may specify configuration file locations in one of two ways:

- By specifying the path to a configuration file.
- By specifying the path to a directory that contains configuration files.

Main and CGI Configuration File Syntax

- Lines that start with `#` are comments and ignored by Nagios.
- Variable names MUST start at the beginning of the line. No whitespace is allowed before them.
- Variable names are case-sensitive.
- Paths in the configuration file may be relative or absolute. Relative paths are relative to the directory where the main Nagios configuration file is located.
- The rules are specific to the MAIN and CGI configuration files.

A Word About Directory Structure

Nagios Core is installed in `/opt/nagios`. As a convenience, we have symlinked `/opt/nagios/etc` to `/etc/nagios`.

Directory structure is important when managing objects in Nagios Core. For very small installations, it's often fine to keep files globbed together in the same directory, but this can lead chaos when managing several score hosts. To keep things organized, we are going to break the `objects` directory into several subdirectorires, one for each type of object we have to deal with.

Object Configurations

Notice that we've created sub-directories for most of our object types in `/opt/nagios/objects`. When managing more than just a few object types, it's wise to break objects into specific categories. How you group these is ultimately up to you; it may be ideal to group objects by location, IP address, function, etc. For now, we're going to group our objects by purpose. We'll build on this structure later in the course.

```
mkdir -v /opt/nagios/etc/objects/commands
mkdir -v /opt/nagios/etc/objects/contacts
```

```
mkdir -v /opt/nagios/etc/objects/contactgroups
mkdir -v /opt/nagios/etc/objects/hosts
mkdir -v /opt/nagios/etc/objects/hostgroups
mkdir -v /opt/nagios/etc/objects/services
mkdir -v /opt/nagios/etc/objects/servicegroups
mkdir -v /opt/nagios/etc/objects/templates
mkdir -v /opt/nagios/etc/objects/timeperiods
mkdir -v /opt/nagios/etc/objects/unused
```

Now move the existing configuration files to their proper locations; we will use these in the main configuration file.

```
cd /etc/nagios/objects
mv localhost.cfg ./hosts
mv contacts.cfg ./contacts
mv commands.cfg ./commands
mv templates.cfg ./templates
mv timeperiods.cfg ./timeperiods
```

After this, the only remaining `*.cfg` files in `/opt/nagios/etc` are unneeded, so we move them into the "unused" directory.

```
mv *.cfg ./unused/
```

System Directory Creation

Log Directory

We'll need to define the location of the Nagios log file directory. Create this directory and make sure the *nagios* user and group have the proper permissions to it:

```
mkdir -v /var/log/nagios
chown -R nagios:nagios /var/log/nagios
chmod -R 0775 /var/log/nagios
```

Runtime Information Directories

We'll create this at `/var/nagios`:

```
mkdir -v /var/run/nagios /opt/nagios/var/perf-data/
chown -R nagios:nagios /var/run/nagios /opt/nagios/var/perf-data/
chmod -R 0775 /var/run/nagios /opt/nagios/var/perf-data/
```

The Main Nagios Configuration File

We're going to create a basic configuration file, and build on it through this course. Move the existing

configuration file for safe-keeping:

```
cd /etc/nagios/objects
mv nagios.cfg nagios.cfg.old
vi nagios.cfg
```

Author the Nagios Configuration File

This is undertaken in the video "Configuring Our Server." When you're done following that video, you should have a *nagios.cfg* file that looks like this:

```
# log files
log_file=/var/log/nagios/nagios.log


# object configuration directories
cfg_dir=/opt/nagios/etc/objects/commands
cfg_dir=/opt/nagios/etc/objects/contacts
cfg_dir=/opt/nagios/etc/objects/contactgroups
cfg_dir=/opt/nagios/etc/objects/hosts
cfg_dir=/opt/nagios/etc/objects/hostgroups
cfg_dir=/opt/nagios/etc/objects/services
cfg_dir=/opt/nagios/etc/objects/servicegroups
cfg_dir=/opt/nagios/etc/objects/templates
cfg_dir=/opt/nagios/etc/objects/timeperiods


# resource file
resource_file=/opt/nagios/etc/resource.cfg


# status
status_file=/var/run/nagios/status.dat
status_update_interval=10


# nagios system user and group
nagios_user=nagios
nagios_group=nagios


# don't log to syslog
use_syslog=0


# read external check commands
check_external_commands=1


# accept passive service and host checks
accept_passive_service_checks=1
accept_passive_host_checks=1
```

```
# strip illegal characters from macro output
illegal_macro_output_chars='~$^&"|'<>


# set default interval length to 60 seconds
interval_length=60

check_external_commands=1
accept_passive_service_checks=1
accept_passive_hosts_checks=1

check_result_path=/opt/nagios/var/spool/checkresults
host_perfdata_command=process-host-perfdata
host_perfdata_file_mode=a
host_perfdata_file_processing_command=process-host-perfdata-file
host_perfdata_file_processing_interval=60
host_perfdata_file_template=[HOSTPERFDATA]\t$TIMET$\t$HOSTNAME$\
t$HOSTEXECUTIONTIME$\t$HOSTOUTPUT$\t$HOSTPERFDATA$
host_perfdata_file=/opt/nagios/var/perf-data/host-perfdata
log_archive_path=/opt/nagios/var/archives
log_rotation_method=d
perfdata_timeout=5
process_performance_data=1
retention_update_interval=5
service_perfdata_command=process-service-perfdata
service_perfdata_file_mode=a
service_perfdata_file_processing_command=process-service-perfdata-file
service_perfdata_file_processing_interval=60
service_perfdata_file_template=[SERVICEPERFDATA]\t$TIMET$\
t$HOSTNAME$\t$SERVICEDESC$\t$SERVICEEXECUTIONTIME$\t$SERVICELATENCY$\
t$SERVICEOUTPUT$\t$SERVICEPERFDATA$
service_perfdata_file=/opt/nagios/var/perf-data/service-perfdata
retain_state_information=1
```

`check_exertnal_commands=1` tells Nagios to check the command file for commands that need to be executed. This is required for the CGIs that power the Web Interface. `accept_passive_service_checks=1` informs Nagios to accept passive service checks initiated by remote hosts, and `accept_passive_host_checks=1` accepts service check preformed by the Nagios Core host.

The `check_result_path` directives tells Nagios Core where to store check results. The `host_perfdata_*` and `service_perfdata_*` directives tell Nagios Core where to store host and performance data, how to structure it, the command to use to process that data, and the processing interval.

The `retain_state_information` directive instructs Nagios Core to retain state information when it shuts down, and read it when it starts up. We want this so we can maintain historical information when we restart the service.

This configuration is basic, but will provide a solid foundation on which to build our understanding of Nagios Core. This will serve us well as we engage additional functionality throughout the course.

All of the directives which can be used in the main configuration file may be found in the "Configuration Directive Definitions" PDF file available as a download for this course. Open or print this file, and look at each of the options we've specified in our configuration file.

Verify Configuration Validity

```
/opt/nagios/bin/nagios -v /etc/nagios/nagios.cfg
```

Plugins

Plugins are standalone programs (scripts or binaries compiled from source) that do the "footwork" of monitoring. Plugins exist in the `libexec` subdirectory of our Nagios installation and can be called from the command line like any other program.

One potential problem with this is that Nagios Core does not really know what a given plugin is monitoring. It's important that plugins adhere to conventions in terms of command-line flags and returning data, but this is also where command objects come into play.

It is wise to acquaint yourself with the plugins themselves. If you change into the `/opt/nagios/libexec` directory, you can view the plugins that are installed with Nagios by default. (They're actually a separate package, but I've installed them on the lab server for convenience.)

As we might expect from any program, the plugins take arguments and parameters. By convention, all plugins must support a standard set of parameters and arguments (though many plugins don't require all of them). We can view this information by executing the plugin from the shell and passing the `-h` or `--help` parameter:

```
/opt/nagios/libexec/check_swap --help
```

The plugin help returns the parameters that are required, are optional, and describe any arguments that may be used. An example is also provided.

A Note About Plugins and Extensibility

Because plugins are programs external to Nagios, they can be used for just about any purpose. The only caveat is that they must accept input and return output according the conventions established by Nagios. This opens a world of opportunities in terms of monitoring. Frustrated because configuring Zabbix to configure free space in MySQL tables is a real kludge? Nagios can do it with a plugin written in just about any language. This makes developing plugins relatively simple and straightforward. Plugin development falls outside the scope of this course, however, and will not be discussed here.

Plugins: How Do They Return Information?

Plugins return information to Nagios Core in a straightforward fashion. The output from plugins is human-readable, as we can see if we run one from the shell:

```
./check_swap -w 50
```

The above command uses the `check_swap` program, and provides the `-w` parameter. If you call this program with `-h`, you see that `-w` takes an integer argument; if the amount of swap available is less than the percentage specified by the integer argument, a warning is generated. Example:

```
SWAP WARNING - 0% free (0 MB out of 0 MB) - Swap is either disabled, not
present, or of zero size. |swap=0MB;0;0;0;0
```

Plugins *always* write their infortmative messages to STDOUT. Plugins return a status, a description, and an exit code. Note that the exit code won't be visible from the shell, but will always match the status:

0: OK
1: WARNING
2: CRITICAL
3: UNKNOWN

Performance data is returned after the status information, following a pipe (`|`) symbol. Performance data may be multiline.

The Nagios Plugin Exchange

There are a wealth of plugins available on the Nagios Plugin Exchange, which is located at: http://exchange.nagios.org/

Understanding Checks

Before we move into authoring objects, it is important to understand how Nagios builds checks. Each check is comprised basically of three different objects as they exist in the relevant definition files:

- Host definitions (which exist in a host's object configuration file) provide host-specific information used to build commands.
- Service definitions are combined with the information provided by the host object; together, they inform Nagios Core which host and what service on that host need to be checked.
- Command definitions not only specify the plugin used for a given check, but also provide a mechanism by which Nagios Core can pass additional information (such as parameters) to the plugin itself.

Hosts, services, and commands, then, are inter-related and integral components of Nagios Core. We are going to take a look at host definitions, command definitions, and macros before we look at the other object

types. The reason for this order will become clear as we move into services and templates.

Active Checks

Active checks are initiated by the Nagios Core host, and with the exception of publicly available ports, typically require an agent like the Nagios Remote Plugin Executor to run.

Passive Checks

These are initiated by remote hosts, and passed to Nagios Core.

Agentless Checks

Agentless checks are those which allow Nagios Core to check a remote host or service without an agent (such as the NRPE) running on the remote host. Some examples of this are:

- Monitoring public ports
- Monitoring via SNMP
- Monitoring host state with ICMP (ping)

Authoring Host Definitions

Hosts are objects we use to describe devices we wish to monitor. These can be anything: virtual machines, network-attached devices like a media center, or run-of-the-mill servers in a data center.

Our host definitions provide Nagios with some critical information: the name, a description, the IP or FQDN, when and how the host should be monitored, and who should be contacted if problems arise.

Host definitions are located in the `/etc/nagios/object/hosts` directory. A host definition must have (at a minimum) the following parameters (also called directives):

- host_name
- max_check_attempts
- check_period
- contacts
- contact_groups
- notification_interval
- notification_period

Examples:

```
define host {
host_name     dbmaster1
alias         db1
address       dbmaster1.somedomain.com
}
define host {
host_name     oracle-server
alias         db
address       oracle-server.somedomain.com
}
```

You may notice that the above examples do not include all of the parameters required. This is because our default configuration makes use of Nagios' templating feature, which we discuss shortly.

A full list of the host definition parameters is found in the downloads section of this course.

Host Groups

Hosts may be grouped together for any number of reasons: by geographic location, for instance, or perhaps by purpose (all web servers, or servers running SSH, etc.)

Hostgroups only require a host group name and alias. There is a PDF which lists all of the Hostgroup Directives in the "Downloads" section of this course.

The Nagios Core Host Definition

The host object for the Nagios Core host (which we are logged in to) is located in the `localhost.cfg` file in `/opt/nagios/objects/hosts/localhost.cfg`.

We are now going to create our own `localhost.cfg`.

```
cd /opt/nagios/objects/hosts
mv localhost.cfg localhost.cfg.old
vi localhost.cfg
```

Make sure your new localhost configuration file looks like this:

```
define host{
name                            localhost
host_name                       localhost
check_period                    24x7
check_interval                  5
retry_interval                  1
max_check_attempts              10
check_command                   check-host-by-ping
```

```
contact_groups                  admins
notifications_enabled           1
event_handler_enabled           1
flap_detection_enabled          1
process_perf_data               1
retain_status_information       1
retain_nonstatus_information    1
notification_interval           120
notification_options            d,u,r
notification_period             24x7  ; this is a comment
}
```

This defines the following:

- The host nickname (the one we see in the UI) is *localhost*
- Host can be found at address *localhost*
- The host should be checked 24 hours a day, 7 days a week
- The host is checked every 5 minutes
- Check retries are spaced 1 minute apart
- Nagios will try, at most, 10 times to check the host
- The command object `check-host-by-ping` is invoked to execute the plugin which performs the check
- This host is part of the Contact Group *admins*; more on that later
- Notifications are enabled for this host
- The event handler for this host is enabled; this means that when the host changes state, pre-configured actions (specified by this directive) can be taken to handle the change of state appropriately
- "Flapping" occurs when a host changes to and from states rapidly. This can be the result of misconfiguration, network problems, etc.; enabling flap detection allows Nagios to avoid excessive alerting in such cases
- Plugins can return data related not only to state, but also performance. The `process_perf_data` directive enables the capture of this information for this host.
- The retaining options allow Nagios to maintain state and non-state related information about the host between restart of the Nagios service.
- The notification interval is specified in units, which are defined by the `interval_length` configuration in `nagios.cfg`. The value specified here indicates how many of these units will be counted between notifications related to this host
- Notification options (see the "Host Definition Directives" PDF file in the "Downloads" section) tell Nagios what states generate notifications: **d** is used for down, **u** for unreachable, **r** for recoveries, **f** for

flapping, **s** for scheduled downtime, and **n** for no notifications at all.

- `notification_period` specifies when notifications for this host can be sent.

Macros

Nagios' macro system provides a wonderful amount of flexibility and reusability. Macros are consistent in definition and use and are context-dependent variables. In addition to the macros defined by default, Nagios provides the functionality to define custom macros.

Macros are used in command definitions and allow information from other objects (hosts, services, etc.) to be referenced dynamically. This reduces the number of commands we have to write. Depending on Nagios' configuration, the built-in macros can also be exported as environment variables. Macros exported as environment variables are named *NAGIOS_[variable-name]* in the plugin's execution environment.

Not all macros are available to every command. Nagios recognizes a number of commands, and certain macros may be available *only* for a specific type of command. Be sure to read the "Macro Definitions" PDF in the downloads for this course.

Note that the *$USERn$* macros are not exported to the environment at all.

Defining Macros

Macros begin and end with a dollar sign ($). For example:

```
define host {
host_name     myserver
address        10.0.0.1
check_command  check-host-alive
}
define command {
   command_name    check-host-alive
   command_line    $USER1$/../libexec/check_ping -H $HOSTADDRESS$ -p 5
}
```

Special Macros

The *$USER1$* macro references the directory which contains the file indicated by the `resource_file` directive in the main Nagios configuration file.

Custom Macros (On-Demand Macros)

Macros can also be customized. They may be used in command definitions and are defined in other object files. Inside the object defining the macro, the directive starts with an underscore and is written in uppercase. For example, in the following host definition, we define the custom macro *_CLUSTERID*:

```
define host {
 host\_name     gallera-1
 address        10.0.0.1
 _CLUSTERID     2
 check_command check-host-by-ping
}
```

In the command definition, the variable can be referenced by prefixing the variable name with the object type, which may be one of the following:

- `$_CONTACT` for variables defined in contact objects
- `$_HOST` for variables defined in host objects
- `$_SERVICE` for variables defined in service objects

So, to reference the *_CLUSTERID* variable defined in our host object above, we use `$_HOSTCLUSTERID` in the command definition:

```
define command {
command_name     check-host-by-ping
command_line     $USER1$/../libexec/check_ping -H $_HOSTCLUSTERID$
}
```

On-demand macros are not exported to the environment. A full list of macros is available as a PDF in the "Downloads" section of this course.

Authoring Service Definitions

Services describe functionality offered by a given host. In the context of Nagios Core, services are not synonymous with daemons such as "SSHD" or "mysqld"; instead, a service object represents any resource that may be observed. For instance: free disk space, TCP errors, InnoDB buffer cache use, etc.

Services are always associated with a host and are identified by their description. As such, services descriptions must be unique on a per-host basis. You cannot have two services with identical descriptions on the same host. You may, however, have services with the same `service_description` directive values, provided the services are all affiliated with separate hosts.

Service definitions exist as part of the service objects defined in the `/etc/nagios/object/service` directory. A service definition must have (at a minimum) the following parameters:

- host_name
- service_description
- check_command

- max_check_attempts
- check_interval_retry_interval
- check_period
- notification_interval
- notification_period
- contacts
- contact_groups

A complete listing of these may be found in the "Service Definition Directives" PDF found in the "Downloads" section of the course.

Service Groups

Service groups, like host groups, can be used to cluster together services. For instance, an "SSH" service group might include all of the SSH services, regardless of host.

Service groups require a host group name and alias. A complete listing of these may be found in the "Servicegroup Definition Directives" found in the "Downloads" section of the course.

Authoring a Service Definition

The first service definition we author is for the Nagios Core Host. We're going to ensure that our system clock is synchronized to the NTP server pools, providing for a reasonable window of variance.

Let's create our service definition file:

```
cd /opt/nagios/etc/objects/services
vi local-time-variance.cfg
```

Our first directive is the host name, which indicates to Nagios which host the service is running on. If you recall, our `localhost.cfg` host file had a *host_name* directive. Because we want to monitor this service on our Nagios Core host, we'll use the name specified in the host file:

```
define service{
 host_name                       localhost
```

Our next directive is the *service_description*, which is used to uniquely identify services on a host. Because we're checking the variance of the local clock from NTP servers, we don't reference NTP or its daemon in the description. While this check is a good determinant of whether or not NTP is functioning properly, we're not monitoring the daemon process directly.

```
service_description     local-clock-offset
```

Now we need to defined the *check_command* directive. We are going to hold off on defining this for now — we delve into commands in a couple of chapters. For the time being, we will include the directive, but not give it a value:

```
check_command           check-local-clock-offset
```

Next, we want to tell Nagios how many times we want to execute this check before determining that it is down. Since we are testing the local system clock (which should be available unless the host is down), we are going to set this value to *1*.

```
max_check_attempts    1
```

Now we want to define the time period in which the service should be monitored. Here we inform Nagios that we can check this service 24 hours a day, 7 days a week:

```
check_period        24x7
```

Next on our list of required directives are our contacts. We have not discussed defining contacts or contact groups, so we will leave these directives undefined for now:

```
contacts
contact_groups
```

Our last two required directives pertain to notification; these are the notification interval and period. We want to be notified if the clock is not in sync once every 8 hours and allow that notification to be sent at any time:

```
notification_interval     480
notification_period       24x7
```

Our `local-time-variance.cfg` file should now look like this:

```
define service{
host_name               localhost
service_description     local-clock-offset
check_command
max_check_attempts      1
check_period            24x7
contacts                nagiosadmin
contact_groups          nagios
notification_interval   480
notification_period     24x7
}
```

Authoring Command Definitions

Commands define how host checks, service checks, and notifications should be performed. Commands inform Nagios as to the specifics of performing a given check, and can be used to check services, run other commands, inform users of issues, and handle events — in short, commands are the glue that ties plugins to Nagios Core.

Unlike other types of objects, commands are short: They have only two directives: a command is defined with the "command_name" directive, and the "command_line" directive. The latter accepts as a value a string that contains the command-line equivalent necessary to call the appropriate plugin.

Event handlers are commands that are executed in response to a change in host or service status.

Building Our First Command

Recall from the previous chapter that we want to check the time offset of the local clock against the time reported by the NTP server pool. We continue that effort here, using one of the plugins provided by Nagios.

Examining Our Plugin and Its Output

If you examine the *check_ntp_time* plugin in `/opt/nagios/libexec` by calling it with the `-h` option, you will see the all of the parameters used in the following command, as well as their meanings.

We're going to test the plugin first by calling it from the shell:

```
check_ntp_time -H 0.us.pool.ntp.org -w 60
```

This command uses the plugin (which is a program) to check the local system clock and compare it to a value retrieved from the NTP servers located at 0.us.pool.ntp.org. A variance of 60 seconds is allowed. If the difference is less than 60 seconds, the returned status will be "OK." If the difference is more than 60 seconds, the returned status will be a warning. We can see what that would look like by passing a value of "0" as the argument to the `-w` parameter:

```
check_ntp_time -H 0.us.pool.ntp.org -w 0
```

The results:

```
NTP WARNING: Offset -0.007536530495 secs|offset=-
0.007537s;0.000000;120.000000;
```

Building Our Command

We are going to build our command definition now. Note that we begin using macros here!

```
cd /opt/nagios/etc/objects/commands
vi check-local-clock-offset.cfg
```

First we define the command name. Notice, that for the sake of simplicity, we're making the command name and the command file the same.

```
define command{
  commmand_name        check-local-clock-offset
```

Now we'll define the *command_line* directive:

```
 command_line         $USER1$/../libexec/check_ntp_time -H 0.us.pool.
ntp.org -W 60
}
```

Remember that the *$USER1$* macro is a reference to the directory in which `resource.cfg` is located.

Adding Our Command Name to Previously Defined Objects

Now that we've defined our command, we need to add its name to the service definition we completed in the previous chapter. Amend the `/opt/nagios/etc/objects/services/local-time-variance.cfg` file, changing the *command_name* directive to match this:

```
 command_name         local-time-variance
```

Notice that the command name directive in the service definition file matches the command name in the command definition file. So your service definition file should look something like this:

```
define service{
host_name               localhost
service_description     local-clock-offset
check_command           check-local-clock-offset
max_check_attempts      1
check_period            24x7
contacts
contact_groups          admins
notification_interval   480
notification_period     24x7
}
```

But My Host Object Already Has a Check Command!

This is okay. Recall that Nagios Core provides us macros which can reference various parts of contact, host, and service objects. This is one reason why — we can use macros to build commands dynamically. This way, there is not a one-to-one relationship between hosts and checks or services and checks. This will

become apparent as we move further into the course. Furthermore, `check_commands` in host definition files are not considered services, but rather the command used to check host state.

Making Commands Reusable and Flexible

Authoring Our Second Command/Using Nagios' Built-in Macros

Recall that when we defined our host, we named a check command: *check-host-by-ping*. That command is as-of-yet undefined, so let's author it now. We're going to make use of a macro in this definition that will keep the command from being tightly-coupled to the host definition. This means this command can be used with a variety of hosts, not just the host definition we have written for the Nagio Core host. Create a file in your command objects directory with the name `check-host-by-ping.cfg`:

```
define command{
command_name          check-host-by-ping
```

Again, if we examine the plugin programs located in `/opt/nagios/libexec`, we see that a plugin is provided for us which allows us to check a host's response via ICMP ping. Run this plugin with the `-h` command to see its various options. The command below will make sense after you do so.

We define our command as follows:

```
check_ping -H localhost -w 2000.00,80% -c 5000.0,100% -p 5
```

Given the above parameters, this plugin returns a warning status if the round-trip time reported is 2 seconds or more or if more than 80% of the ICMP packets are lost. If the round-trip time is 5 seconds (or more) or 100% of the ICMP packets are lost, a critical status is returned. A total of five pings will be sent.

Let's create a command definition that we can use to check our hosts by ping, but rather than hard-coding a value for the host (the `-H` parameter), let's use a macro to make this command more flexible.

Open `/opt/nagios/etc/objects/commands/check-host-by-ping.cfg`. Again, we'll create a command definition, and provide a name for the command (the one previously specified in our host definition):

```
define command{
 command_name          check-host-by-ping
 command_line          $USER1$/../libexec/check_ping -H $HOSTADDESS$ -w
2000.00,80% -c 5000.0,100% -p 5
}
```

This accomplished two things. First, this command can now be used with any host object, as the host passed to the *check_ping* plugin is a macro. Nagios will replace the *$HOSTADDRESS$* string with the host address of the host object being used. Secondly, because the host address is not static, we can use this command with any host object and can be assured that it will behave as we expect it to.

Passing Parameters to Commands

So we've made our *check-host-by-ping* command reusable by employing one of Nagios' built-in macros. But there are still some limitations. For instance, it may not be appropriate to issue a warning if 80% of our ICMP packets are dropped. For a high-availability system on the same subnet, we probably don't want ANY of our ICMP packets to go unacknowledged.

Now, we could write another command for this. In a small installation, that will not pose a huge problem. But what if you plan for your Nagios installation to grow? To support tens, scores, maybe hundreds or thousands of devices? Writing "one off" commands is a quick way to make Nagios somewhat unmanageable. If you have to write a new command for every variation on ICMP ping that you happen to need, you could end up with hundreds of similarly named commands, and that's just not ideal. Or easy.

To avoid this scenario, Nagios allows us to parameterize commands. So, instead of hard-coding the values for the `-c` and `-w` flags in our *check-host-by-ping* definition file, we can use use special macros to pass these values from the host or service object that calls our command.

Arguments are passed to commands by prefixing each with an exclamation mark following the command name and then specifying the argument value. Additional arguments can be added with another "!\<value\>" pairing.

For example, in our `localhost.cfg` file (where we call the *check-host-by-ping* command object), we can pass arguments to our command by altering this line:

```
check_command                  check-host-by-ping
```

So that it looks like this:

```
check_command                  check-host-by-
ping!2000.00,80%!5000.00,100%
```

Notice there are two arguments:

```
!2000.00,80%
!5000.00,100%
```

These are available in our command definition as *$ARG1$* and *$ARG2$*, respectively. Let's amend our our `check-host-by-ping.cfg` file to use the arguments passed to it. We're going to change this line:

```
command_line    $USER1$/../libexec/check_ping -H $HOSTADDRESS$ -w
2000.00,80% -c 5000.00,100%
```

To this:

```
command_line    $USER1$/../libexec/check_ping -H $HOSTADDRESS$ -w $ARG1$
-c $ARG2$
```

So now instead of having to write two different command definitions when we need different values for the ICMP round-trip thresholds and the threshold for the number of dropped ICMP packets, we can simply specify these anywhere we name the *check-host-by-ping* command.

These macros can be used with any command, and they are one of Nagios most powerful features.

Don't Forget!

Remember to look PDF in the downloads section for a complete list of macros. And don't forget to investigate all of the plugins provided with Nagios in the `libexec` subdirectory of our Nagios installation.

Authoring Time Period Definitions

Time periods, in the context of Nagios, describe ranges of days, dates, and times. These can be used to specify when specific actions may be undertaken, when notifications may be sent, and when certain contacts or groups of contacts may be notified. (We haven't discussed contact or notifications yet — but we're almost there!)

Dates and Days

Time periods are very flexible in Nagios. Dates, in particular, can be specified a number of ways; this information may be found as a PDF in the downloads section of this course.

Note that the above items are in order of execution; that is, items using a date form higher on the list are executed BEFORE items using a date form that is lower on the list.

Time Periods

Time periods are Nagios' way of letting us define at what hours and days an action, such as a check, notification, or a specific command, can be undertaken. Time periods consist of a list of days and a time.

Times are specified using the 24-hour clock, so "5:00 PM" is represented as "17:00". Here's an example definition of a time period we've already used and so should be familiar with:

```
define timeperiod{
 timeperiod_name  24x7
 alias            24 hours a day, 7 days a week
 monday           00:00-24:00
 tuesday          00:00-24:00
 wednesday        00:00-24:00
 thursday         00:00-24:00
```

```
 friday            00:00-24:00
 saturday          00:00-24:00
 sunday            00:00-24:00
}
```

Here's another time period, which uses a variety of methods to indicate when events may occur:

```
define timeperiod{
 2017-01-01 - 2017-01-30 / 2     15:00-17:00 ; Every three days from
                                             ; from January 1, 2017
                                             ; through January 30,
                                             ; but only between 3
                                             ; and 5 PM
 monday 3 - thursday 4           00:00-24:00  ; 3rd Monday to 4th
                                             ; Thursday of every
                                             ; month
day 1 - 15                       00:00-24:00  ; 1st to 15th day of
                                             ; every month
day 20 - -1                      00:00-24:00  ; 20th to the last day
                                             ; of every month
}
```

Contacts

Contacts are one of the object types that are integral to Nagios. Contacts provide information to Nagios about who should be notified, what for, and when.

Nagios has provided a basic `contacts.cfg` file for us, which is located at `/opt/nagios/etc/objects/contacts/contacts.cfg`. Let's take a look at `contacts.cfg`. Most of the directives are self-explanatory, except one. The *use* directive is one we've seen before, but we haven't used. (We'll get to that soon, when we cover templates.)

Apart from that, however, contacts and contact groups are defined just as we've defined our other objects. The biggest difference between contacts and other objects, however, is that contacts define who can log in to the Nagios web interface. (The login password has to be set using the "htpasswd" tool, but we'll deal with that when we discuss configuring Apache.)

We are going to alter the contents of the contacts file to make it more readable, which really means removing the comments and the extra whitespace. Our final file should look like this:

```
define contact{
contact_name                    nagiosadmin
alias                           Nagios Admin
email                           your_email_address@here.com
service_notification_period     24x7
host_notification_period        24x7
service_notification_options    w,u,c,r,f,s
host_notification_options       d,u,r,f,s
```

```
service_notification_commands   notify-service-by-email
host_notification_commands      notify-host-by-email
}

define contactgroup{
    contactgroup_name       admins
    alias                   Nagios Administrators
    members                 nagiosadmin
}
```

All of the directives which may be used in the generation of contacts may be found in the "Contact Definition Directives" file in the "Downloads" section of this course.

Verifying Our Configuration

We can use the Nagios binary itself to verify the validity of our configuration:

```
/opt/local/bin/nagios -v ...
```

Note that verification only works when run against the main configuration file.

Nagios will verify the configuration and report any errors it finds. We will find quite a few on our first verification attempt; it is to be expected. It is wise to check run the configuration check after every change is made; this can avoid unnecessary downtime with the Nagios service itself.

Permissions

We need to ensure that the entire `/opt/nagios` directory is owned by the *nagios* user. The objects directory and its contents need to be mode 755 for the WebUI to operate correctly, also.

Starting Nagios

We want Nagios to run at boot. If we look at the currently-installed unit files, we can see that Nagios is not currently installed:

```
systemctl list-unit-files | grep nag
```

It does not appear that Nagios is installed as a legacy service either, which we can verify with the `chkconfig` command:

```
chkconfig --list
```

We can remedy this by installing Nagios using systemd; enable Nagios with the following command:

```
systemctl enable nagios.service
```

Now if you run `chkconfig` you'll see the *nagios* service is listed. It should be enabled for run levels 2, 3, 4, and 5.

Starting and Stopping Nagios

You can start and stop Nagios as you would any systems-capable daemon, using the stop/start/restart commands. If you're not as familiar with systemd, the `service` command accomplishes the same thing.

Go ahead and start Nagios; check the log files and the system journal for any errors that might be encountered.

Notifications

Notifications are provide contact information regarding hosts, services, and their status. When and how notifications are sent is determined by the directives in the contact configuration.

Notifications are typically sent out when:

- A host's state changes to DOWN or UNREACHABLE
- A host or service remains in the DOWN or unreachable state, in which case a notice is sent according to the value set in the "notification interval" directive
- A host or service recovers and the status is returned to "UP" or "OK"
- The host or service starts or stop flapping or remains flapping

When host or service changes state, Nagios' logic begins determining what information needs to be sent to whom. That logic (at its most basic) follows this pattern:

- The host or service condition must change
- The date and time of the event are checked to ensure the current date and time fall within a range specified by a "notification time period" directive

Only if these two conditions are true will Nagios generate a notification. The users and groups linked to the host or service are enumerated; the user and group objects again are compared to a time period object to determine if notification is allowed.

Notification Filters

There are a number of filtering mechanisms which give Nagios fine-grained control over whom gets

notifications:

- **System-wide notifications** • This is a singular setting and is located in the main Nagios configuration file. It will enable or disable notification for the Nagios Core Host and all of the devices and services it monitors.
- **Host filtering** • The "notification options" directive determines whether or not notifications related to the host in question will be sent.
- **Service filtering** • Identical to the host filtering option above, but pertain only to the service in question.
- **Host and service groups** • Identical to their singular counterparts, save that they act on groups of hosts and services.
- **Contact filtering** • Nagios makes use of contact groups to allow you to define whom gets contacted about which events, and when. This includes "multi level" notifications, where the same contact receives messages from the same host or service for two different states.

Configuring Apache

For us to use the WebUI included with Nagios by default, we need to install a couple of software packages. First, let's make sure that our installations of Apache and PHP are unaltered:

```
yum reinstall httpd php
```

Once that has completed, let's install some necessary dependencies:

```
yum install openssl mod_ssl mod_php
```

At this point, you shouldn't have to make any other changes to the Apache or PHP configuration in order for the Nagios WebUI to function properly. he above installs will enable the PHP and SSL modules in Apache, and conveniently (on CentOS 7), a self-signed certificate will be generated which we can use to protect the WebUI from prying eyes.

Nagios Configuration for Apache

When the Nagios Core software is installed, an Apache configuration file is placed in `/etc/httpd/conf.d/nagios.conf`. If you're missing this file for some reason, make sure that it contains this information:

```
ScriptAlias /nagios/cgi-bin "/opt/nagios/sbin"

<Directory "/opt/nagios/sbin">
   SSLRequireSSL
   Options ExecCGI
```

```
    AllowOverride None
    <RequireAll>
      Require all granted
      AuthName "Nagios Access"
      AuthType Basic
      AuthUserFile /opt/nagios/etc/htpasswd.users
      Require valid-user
  </RequireAll>
  AuthName "Nagios Access"
  AuthType Basic
  AuthUserFile /opt/nagios/etc/htpasswd.users
  Require valid-user
</Directory>

Alias /nagios "/opt/nagios/share"

<Directory "/opt/nagios/share">
   SSLRequireSSL
   Options None
   AllowOverride None
   <RequireAll>
     Require all granted
     AuthName "Nagios Access"
     AuthType Basic
     AuthUserFile /opt/nagios/etc/htpasswd.users
     Require valid-user
  </RequireAll>
  AuthName "Nagios Access"
  AuthType Basic
  AuthUserFile /opt/nagios/etc/htpasswd.users
  Require valid-user
</Directory>
```

Permissions in the Nagios Core CGI Configuration File

Make sure that the cgi.conf file located in /opt/nagios/etc has the following directives set as indicated:

```
main_config_file=/opt/nagios/etc/nagios.cfg
physical_html_path=/opt/nagios/share
url_html_path=/nagios
show_context_help=1
use_pending_states=1
use_authentication=1
use_ssl_authentication=0
authorized_for_system_information=nagiosadmin
authorized_for_configuration_information=*
authorized_for_system_commands=nagiosadmin
authorized_for_all_services=*
authorized_for_all_hosts=*
authorized_for_all_service_commands=nagiosadmin
authorized_for_all_host_commands=nagiosadmin
default_statusmap_layout=6
default_statuswrl_layout=3
```

```
ping_syntax=/bin/ping -n -U -c 5 $HOSTADDRESS$
refresh_rate=10
result_limit=100
escape_html_tags=0
action_url_target=_blank
notes_url_target=_blank
lock_author_names=1
navbar_search_for_addresses=1
navbar_search_for_aliases=1
```

File Permissions

The directory containing the web content and CGIs used to generate the WebUI must be accessible to the Apache daemon. The CGIs must be executable, and care should be taken to with security both in Apache and on the filesystem to ensure the CGIs are not compromised.

System Group Membership

To provide Apache and the CGIs access to the Nagios log files, we need to add the user under which Apache runs to the "nagios" group. The same goes for handling commands passed to Nagios by the WebUI; we need to ensure proper group membership. We do that with the following commands:

```
usermod -a -G apache nagios
usermod -a -G nagcmd apache
usermod -a -G nagcmd nagios
```

MIME Types for CSS and JavaScript

Certain versions of Apache may not correctly detect the MIME type for CSS and JavaScript files. To fix this, we need to add the two directives to the end of `/etc/httpd/conf/httpd.conf` file:

```
AddType text/css .css
AddType text/javascript .js
```

Starting Apache

Once the necessary software is installed, Apache can be started using systemd, and should be automatically started at boot. Make sure systemd starts Apache by issuing the following command:

```
systemctl enable httpd.service
```

The Apache Configuration Files in the Nagios Core Software

There are some changes we need to make to the Nagios Core configuration for the WebUI to function as

we'd like.

User Permissions Within Nagios Core

Nagios' default settings provide users access *only* to the services they are responsible for by default. We use Apache to handle access and authentication to the WebUI, but Nagios Core itself handles access to various objects.

It is very important to make sure that contact names (in contact definitions) match the user names stored in the Apache user database; if there is a mismatch, Nagios Core will not correctly map the Apache database used to the contact. If this occurs, the user (contact) will not be able to see all of the necessary information in the WebUI.

That said, let's take a look at the CGI configuration on our Nagios Core host and examine the directives within. We'll use the default configuration as opposed to authoring our own, since the default configuration makes use of values we might use in a file we author ourselves.

The Apache User Database

As mentioned in the Apache configuration snippets previous, our Apache user database is located at `/opt/nagios/etc/htpasswd.users`. We manage users with the `htpasswd` utility provided with Apache.

One reason for allowing Apache to manage access and authentication to the WebUI is that is Apache is fairly modular and has modules designed to fit this exact task. In the context of this course, however, we're going to stick with the simplest configuration, which is to use the `htpasswd` utility to manage users.

First, let's create an administrative user. Note that you only want to use the `-C` flag when creating the password database the first time.

```
htpasswd -C /opt/nagios/etc/htpasswd.users nagiosadmin
```

You will be prompted to provide a password for this user. Once you've done that, visit the Nagios WebUI in a web browser. The URL you use will be:

```
http://<your.serverlab.fqdn>/nagios
```

When prompted for credentials, enter the user you created previously with the `htpasswd` utility. Provided you keep Apache users in match with Nagios contacts, this is how additional users can be added to the system.

It is possible to create administrative users with no contact information at all by specifying their Apache usernames in the `cgi.cfg` file. If a user in the Apache password database does *not* match that of a Nagios contact, be aware the the user in question may not be able to see much information in the WebUI (unless they are given administrative privileges in the *cgi.cfg* file). Nor will that person receive notifications of any

kind, as no corresponding contact information exists.

The Nagios Web User Interface (WebUI)

Nagios Core's WebUI is the primary means of obtaining information in real-time about hosts and services.

Login

You should be able to access the Nagios WebUI at: `https://<yourserver.fqdn>`

The login is the same we created previously, using the `htpasswd` command.

Interfaces

In addition to the default interface provided with Nagios Core, a number of interfaces can be found on the Nagios Plugin Exchange. Nagios V-Shell is a PHP-based interface which can be installed in place of the default WebUI. V-Shell is expected to become the default Web User Interface at some point in the future.

Reports

Availability Report

The availability report provides historical uptime for hosts and services. The information provided can be useful for obtaining compliance information or (for example) if SLA requirements are being met.

The availability report may be one of four types, as selected in Step 1:

- Hostgroup
- Host
- Servicegroup
- Service

In Step 2, the objects for which the report is generated can be narrowed; you may select "ALL" specified hosts/hostgroups/services/servicegroups.

Step 3 allows you to specify the time period covered by the report; the first option provides default time periods.

There are additional options which may be selected; see the "Report State Options" section below.

Trends

The these reports provide a graphical timeline breakdown of the state of a particular host or service. The options are nearly the same as those of the Availability Report.

Alerts

Alert History

The alert history report provides a record of alerts for hosts and services. The report may be filtered using the options at the top-right of the window.

Alert Summary

This report uses the same options as the Availability Report, but provides information regarding the hosts and services which produce the most alerts. This can be useful in terms of identifying problematic hosts and services on your network.

Notifications

Like the Alert History report, this report provides a hitorical record of host and service notifications. The contacts to which notifications are sent are also listed. The report may be filtered using the options at the top-right of the window.

Report State Options

Many of the reports will allow you specify parameters beyond the object type and time period. These options are described below.

- **Assume Initial States** • If this is selected, Nagios Core will not assume the state of the host or service if the log lacks this information, or if the log is absent. In this case, the initial state will be taken from the "First Assumed State" field, if present.
- **Assume State Retention** • If Nagios Core is restarted, checking this option provides for the assumption that they are in the same state they were in prior to the reboot. Don't use this without setting the "retain state information" directive in the main configuration file.
- **Assume States During Program Downtime** • If Nagios Core finds it was down during the reporting period, it will assume the final states (before the period of downtime) for hosts and services.
- **Include Soft States** • Do not include/include soft states in the report.
- **First Assumed State** • This is the state Nagios Core will assume for hosts or services if the actual state can't be determined from the log files.
- **Backtracked Archives** • This shows the number of archived log files Nagios Core will process to determine initial states.
- **Suppress Image Map** • This suppresses popups.

Host Groups

So far, we've more or less set aside Host Groups, Contact Groups, and Service Groups. This is one area where Nagios Core's flexibility can also be its Achilles' heel: groups which are ambiguous in purpose, poorly defined, or sloppily managed can quickly break a Nagios installation. At best, poorly-planned configurations may result in Nagios being down. At worst, contacts may received endless notifications about problems which may not truly exist.

Planning is key.

Host Group Membership Management Strategies

If you look at the "Host Definition Directive" and "Hostgroup Definition" directive PDFs in the Downloads section of this course, you will notice the *host_groups* and *members* directives in their respective files. If you're thinking that these two directives provide similar functionality, you're correct. You may place a host in a host group by specifying a comma-separate list of host groups in the host definition.

You may also place a host in a host group by placing the host name (the *host_name* directive) in the hostgroup definition.

So why the duplicate functionality? One answer is that it depends on how you manage your hosts and host groups. For small installations, for instance, it may be more efficacious to manage groups by specifying them for each host. This has the advantage of keeping all of the host-related information in the host definition. On the other hand, may mean that you have to alter a number of host files when a host group is created, changed, or deleted.

Conversely, you may manage host group membership using hostgroup definitions. This may be more well-suited to large installations, where host group members are enumerated by the tens, scores, or greater.

Whichever route you choose, keep in mind that both directives apply: a host can be placed in a host group by either directive.

Authoring Our First Hostgroup

Create a *hostgroups* directory in `/opt/nagios/etc/objects` and make sure the owner and permissions are set correctly. Change into the newly created directory, and open our first host group definition file:

```
vi linux-academy-servers.cfg
```

Make sure the contents match the following:

```
define hostgroup{
        hostgroup_name     linux-academy-servers
    members             localhost
```

```
}
```

Right away there is a problem with our configuration. We've defined the host definition for our Nagios Core host using the host name *localhost*. This is ambiguous, because it does not actually tell us WHICH host the localhost is. Let's alter our host configuration and change the host name to *nagios-core-host-1*. We're including the "1" here as a best practice; we may have to expand our installation later, so it is wise to have a naming convention which accommodates growth in place.

Let's also add an alias to our host definition:

```
    alias     nagios-core-host-1
```

Save and exist the host definition file.

Now, in keeping with best practices, let's now change the name of our host definition file so it reflects the host name:

```
mv localhost.cfg nagios-core-host-1.cfg
```

Last, let's go back to the host group definition file we created, and change the members directive to reflect the new name of our host:

```
define hostgroup{
        hostgroup_name     linux-academy-servers
    members             nagios-core-host-1
}
```

Verify the configuration, fix any errors, and restart Nagios. The host group should now be visible in the WebUI.

Notes

Keep in mind we could have defined the host group without a members directive, and used the *host_groups* directive in the host definition to place the host in our new host group.

Hostgroups may include other host groups using the *hostgroup_members* directive. This will include all members of the referenced host group in the group being defined.

Host group names must be unique system-wide.

To exclude a specific host from a host group, prefix the host name with an exclamation mark (!). *Use this option with caution; it lends itself to confusion.*

Service Groups

Service definitions can be used to group services together. This is useful particularly when you have several hosts running the same service. For example, if you have a load-balanced website running in EC2, grouping the Apache service into a service group will allow you to immediately see the status of that service across all hosts.

Service Group Membership Management Strategies

We are going to take a different approach to managing service group membership. Unlike host group membership, where we're placing the hosts in the *members* directive of the host group definition file, with the service groups, we're going to specify group membership in the service definition.

Authoring Our First Service Group

Let's author our first service group definition:

```
vi local-clock-offset-service.cfg
```

Make sure the definition looks like this:

```
define servicegroup {
     servicegroup_name     local-clock-offset-service
     alias                 Local Clock Offset from NTP Check
}
```

And now let's edit the service definition and add the service to this group:

```
     servicegroups     local-clock-offset-service
```

Verify the configuration, fix any errors, and restart Nagios. The service group should now be visible in the WebUI.

Contact Groups

We've already written our contact group definition, but take the time to review it, as well as the **Contact Definition Directive** and **Contactgroup Definition Directive** reference PDFs available in the downloads for this course.

Templates

If you've compared any of the hosts, contacts, or services we've written to the examples provided by the Nagios Core software, you'll notice some big differences. The default examples make use of templating, something we purposely avoided so as to familiarize ourselves with as many of the object-specific directives as possible. This has the added benefit of keepings things simple while we learn what the various aobjects are and how they fit together in Nagios Core.

That accomplished, however, one thing is immediately apparent: object definitions share many directives in common. Without templates, we'd have to include these directives in every definition. This is not only tedious but lends itself to error.

Templates help us avoid both the tedium and the tendency toward error by allowing us to define templates which various objects can "use." An object which uses a template inherits the directives in the template and does not have to redefine those directives in its definition.

Template Types

Templates are defined identically to other objects. They are recognized as templates (and not as objects) when the *register* directive is assigned a value of "0" in the definition file.

To define a very simple host template, for example:

```
define host{
    name                servers
    check_interval      5
    register            0
}
```

The above template, then, when "used" by another host template or host definition, will provide the "using" template or definition the above directives and their corresponding values. Values will be overridden only if the "using" template or host definition redeclares the directive.

Unlike object definition files, template definitions DO NOT have to contain all of the required directives for a particular object type.

Inheritance

Templates can include other templates. For instance, given the following template:

```
define host{
    name                check-five-times
    use                 servers
    max_check_attempts  5
    register            0
}
```

This template, then, inherits the directives provided in the "servers" template previously defined. Any

templates or host objects which "use" the "check-five-times" template will, by default, include all of the directives provided in both templates.

REMEMBER: Directives in templates are overridden if they are redefined in the current object or template definition file.

Other Template Types

Templates may be created for contacts, hosts, and services.

The Special Case of Timeperiods

Timeperiods contain a special *exclude* directive that acts like the *use* directive in other objects with one major difference: the time period specified by the *exclude* directive will be *removed* from the current time period definition.

Replacing Our Object Tree

When we set out to learn Nagios Core, removing the defaults provided by Nagios was ideal so we could learn the in-and-out of defining objects, object groups, templates, and macros.

Now that we understand templates and macros in particular, though, it will be helpful to us to restore those defaults. These are provided in the Downloads section of this course in a file named *objects.tar.gz*. Download this to your computer, and SFTP it to your Nagios Core host.

Copy the file from the directory where it is uploaded to `/opt/nagios/etc`. Then execute the following commands to replace the our configuration:

```
rm -rf /opt/nagios/etc/objects
gunzip objects.tar.gz
tar -xvf objects.tar
```

We'll want to be sure that the new objects directory has the right ownership and permissions, so execute the following commands:

```
chown -R nagios:nagios /opt/nagios/etc/objects
chmod -R 755 /opt/nagios/etc/objects
```

Verify your configuration:

```
/opt/nagios/bin/nagios -v /opt/nagios/etc/nagios.cfg
```

Take some time to browse through the objects directory. We've put the original Nagios commands, services, and templates into place. You'll see this reflected in the WebUI after you restart the Nagios service.

Adding Remote Hosts

We're going to begin adding additional hosts to our Nagios Core setup now. First, though, we need to do a little bit of preparation. Let's consider a few things:

- What groups do these hosts belong in? However you choose to break your hosts into groups, make sure new hosts are assigned properly.
- What services run on new hosts? Again, make sure service groups are assigned properly.
- Ensure contacts (if needed) are added so that notifications are sent to the right contacts.
- Add contact groups, for the same reason.

Maintain conformity with in terms of established object names.

Spinning Up Additional Servers in Server Labs

Spin up a couple of new servers in Server Labs. Make note of the FQDN of each.

Defining the Hostgroup

Now we've got one group for Linux Academy Servers established and another for localhosts. Let's add a third group to that: remote hosts. Be sure to double-back and place the new host in this group; for the time being, leave the "members" directive blank.

Create the Service Group

Create a new servicegroup in the file `http-80-service.cfg`. This is where we'll put services which serve http over port 80.

Create the Host Configuration

Copy the existing configuration for the Nagios Core Host to `<fqdn-of-your-serverlabs-server>.cfg`. Edit this file, replacing the alias, name, host_name, and display_name directives with the FQDN of your server.

Look at the WebUI

Validate the Nagios configuration and visit the WebUI. You should see the new host in the appropriate host groups. As there are no services configured on the new host, you won't see services.

Make sure the new host is in the appropriate host groups.

Installing the Nagios Remote Plugin Executor

The Nagios Remote Plugin Executor, or NRPE, is a plugin which allows the Nagios Core host to execute plugins on remote hosts. The plugins must be installed on the remote host locally for this to work. They do not necessarily have to be the same plugins installed on the Nagios Core host.

Compiling and Installing

Since the image we're using has been "slimmed down" (unnecessary packages have been removed to save space), we need to ensure the necessary software to compile is properly installed. Several requisite programs and sets of development headers may be missing.

Execute the following steps to prepare the system for compilation:

Reinstall the development tools:

```
yum group mark remove "Development Tools"
yum groupinstall "Development Tools"
```

Install dependencies:

```
yum install krb5-devel openssl-devel
```

Now obtain the NRPE source, which can be found at exchange.nagios.org. Search for NRPE. The latest version at the time of writing is 3.0.1.

Download the `.tar.gz` file into `/root`:

```
gunzip <filename>tar.gz
tar -xf <filename>.tar
cd <filename>
```

Configure the source package with the following command. If you're not using the image provided in the Server Labs, you'll need to determine the appropriate options for your distro.

```
--prefix=/opt/nagios --exec-prefix=/opt/nagios --enable-ssl --enable-
command-args --enable-bash-command-substitution --with-opsys=linux
--with-dist-type=rh --with-init-type=systemd --with-nrpe-user=nagios
--with-nrpe-group=nagios --with-nagios-user=nagios --with-nagios-
group=nagios
```

Build the software with the `make` command:

```
make && make all && make install && make install-config && make install-
init
```

Look for the *check_nrpe* executable in `/opt/nagios/libexec`. If it's not present, something's wrong.

Configure NRPE

Before we edit the NRPE configuration, we need the IP address of the Nagios Core host as reported from the `ifconfig` or `ip add` command on the command line.

Open the `/opt/naios/etc/nrpe.cfg` file. We're going to change the following options:

```
debug=1
allowed_hosts=127.0.0.1,<interal IP of Nagios Core Host>
dont_blame_nrpe=1
allow_bash_command_substitution=1
allow_weak_random_seed=1
ssl_client_certs=0
```

Enable the NRPE Service

```
systemctl enable nrpe.service
```

Make sure the NRPE process is running:

```
ps ax | grep nrpe
```

And check with systemd as to the status:

```
systemctl status nrpe.service
```

NRPE should now be installed and will start at boot.

Executing Checks With the NRPE

To continue with this course, you'll need to have successfully completed all of the exercises after the "Nagios: The Basics, Part 1" section.

Why the NRPE?

The NRPE is the preferred method for executing remote checks, as it requires less overhead than SSH in terms of system resources, as well as less overhead in terms of systems management. Using SSH requires

the management of SSH keys system-wide, which presents another layer of administrative work. For these reasons, the use of the NRPE is preferred.

NRPE Commands

The NRPE config file makes use of the "include_dir" directive in a manner consistent with the main Nagios Core configuration file.

Two things are very important here:

- NRPE must have been compiled with support for command arguments. If not, the commands it registers won't be able to process command arguments, and you'll be limited to hard-coded values specified in the NRPE config file. With command argument processing specified, you can call the command from a service definition with arguments, which provides much greater reusability and flexibility.
- The "don't blame nope" directive in the NRPE configuration file must be set to "1" to process commands with arguments.

That said, all of the commands available to the NRPE must be provided in the NRPE configuration file. We're going to include the following commands at the bottom of our NRPE configuration:

```
command[check_users]=/opt/nagios/libexec/check_users -w $ARG1$ -c $ARG2$
command[check_load]=/opt/nagios/libexec/check_load -w $ARG1$ -c $ARG2$
command[check_disk]=/opt/nagios/libexec/check_disk -w $ARG1$ -c $ARG2$
-p $ARG3$
command[check_procs]=/opt/nagios/libexec/check_procs -w $ARG1$ -c $ARG2$
-s $ARG3$
```

NRPE Commands on the Nagios Core Host

Back on the Nagios Core host, we'll need to write additional commands. This is because we don't call commands on remote hosts directly, but instead call them using the `check_nrpe` plugin. Take a look at that plugin's help options using the `-h` flag. Note in particular, the `-c` and `-a` flags. These allow you to specify the command name on the remote host, and the arguments that will be passed along from the command definition on the Nagios Core host to the remote host.

We'll now define a single file for all of our general remote-host commands (specific to Linux) in `/opt/nagios/etc/objects/commands/remote-linux-commands.cfg`:

```
define command{
        command_name    nrpe-check-disk
        command_line    $USER1$/../libexec/check_nrpe2 -H
"$HOSTADDRESS$" -c check_disk -a 20% 10% /data
        }

define command{
        command_name    nrpe-check-load
```

```
        command_line    $USER1$/../libexec/check_nrpe2 -H
"$HOSTADDRESS$" -c check_load -a 5.0,4.0,3.0 10.0,6.0,4.0
        }

define command{
        command_name    nrpe-check-procs
        command_line    $USER1$/../libexec/check_nrpe2 -H
"$HOSTADDRESS$" -c check_procs -a 250 400 RSZDT
        }

define command{
        command_name    nrpe-check-users
        command_line    $USER1$/../libexec/check_nrpe2 -H
"$HOSTADDRESS$" -c check_users -a 5 10
        }

define command{
      command_name    nrpe-check-swap
      command_line    $USER1$/../libexec/check_nrpe2 -H "$HOSTADDRESS$"
-c check_swap -a 10% 5%
    }
```

Notice that the command name specified with the `-c` flag matches, EXACTLY, the command name on the remote host. Verify your configuration is valid and troubleshoot any errors before continuing.

The NRPE configuration file must contain all of the commands you call remotely from the Nagios Core host using the `check_nrpe` plugin!

Writing Services for Remote Hosts

Adding a New Service Template for Remote Hosts

Before we write the services for out remote hosts, let's amend our templates so we can define some useful defaults. In `/opt/nagios/etc/objects/templates/service.cfg`, add the following after the `local-service` template definition:

```
define service{
    name                            remote-service
    use                             generic-service
    max_check_attempts              5  ; this # checks
                                       ; before state goes from "SOFT"
                                       ; to "HARD"
    normal_check_interval           2
    retry_check_interval            1  ; Re-check the service every
                                       ; minute until a hard state can
                                       ; be determined
    register                        0
    }
```

Verify your configuration is valid before continuing.

Adding Remote Services

Now that we've got a template to use, let's get busy writing some services for out remote hosts. In the `/opt/nagios/etc/objects/services` directory, create a new file called `remote-services.cfg`, and add the definitions below:

```
define service{
    use                             remote-service
    hostgroup_name                  remote-hosts
    service_description             Root Partition
    check_command                   nrpe-check-disk
}

define service{
    use                             remote-service
    hostgroup_name                  remote-hosts
    service_description             Current Users
    check_command                   nrpe-check-users
}

define service{
    use                             remote-service
    hostgroup_name                  remote-hosts
    service_description             Total Processes
    check_command                   nrpe-check-procs
}

define service{
    use                             remote-service
    hostgroup_name                  remote-hosts
    service_description             Current Load
    check_command                   nrpe-check-load
}

define service{
    use                             remote-service
    hostgroup_name                  remote-hosts
    service_description             Swap Usage
    check_command                   nrpe-check-swap
}
define service{
    use                             remote-service
    hostgroup_name                  remote-hosts
    service_description             SSH
    check_command                   check_ssh
    notifications_enabled           1
}
```

Make sure the file ownership, group, and permissions are set appropriately. Verify your configuration after adding this file.

Hostgroup Assignment

For the services above to be run against our newest host, it will have to be placed in the proper host group definition file. Go to the "Assigning Host to a Hostgroup" exercise to complete this step.

Troubleshooting Problems with the NRPE

You'll notice that some of the checks being executed with the NRPE are encountering an error. This is likely because we've used the FQDN of our lab servers when we need to specify the private IP address as provided in the Server Labs screen. Go back and add the "address" host definition directive to each of our remote hosts, specifying the internal IP address. We have to do this because the FQDN will resolve to the external IP of the lab server, and we can't listen directly on that interface.

We also may need to install and configure the NRPE on other hosts as we move forward, so be sure to double check that the NRPE service is up and running on each of the remote hosts that we are monitoring.

Passive Checks

Passive checks must be enabled in the main Nagios configuration file; these two options are self-explanatory:

- accept passive service checks
- accept passive hosts checks

Service configuration is identical, save that you must use these two options in the service definition:

- active checks enabled
- passive checks enabled

Explanation of these directives is available in the "Service Definition Directives" PDF available in the downloads section.

Active Checking After Period of Non-Reporting

If a service which reports passively stops reporting, Nagios Core can be configured to use active checks to obtain the status of the service in question. To do this, we need to set the "active checks enabled" directive to a value of "1," and without specifying the "normal check interval" directive (locally or in a template).

The "check freshness" is best set to "1," and "freshness threshold" should be set to the maximum non-reporting tolerance, in seconds. "check freshness" tells Nagios to check whether or not results are up-to-date.

Local vs. Remote Passive Checks

Passive checks on the Nagios Core host can write their output to the command file - the same used by the CGI's, which schedule active checks for the WebUI.

Passive checks which are executed remotely have to be transported to the server across the network, which means a plugin like the Nagios Remote Data Processor has to be used to allow the checks to be submitted.

It is best to avoid using the Nagios Service Check Acceptor, as it has numerous security vulnerabilities and hasn't been updated in more than five years.

When to Use Passive Checks

Passive checks are best used only when active checks cannot be used. For instance, when an active check is blocked by firewall rules, or when the NRPE plugin daemon cannot be run on the remote host. Passive checks can also be used to send information to Nagios Core when check results take longer to obtain than is reasonable with an Active Check. Calculating per-table free space in MySQL, for instance, is one good example.

Event Handlers

Event handlers are identical to commands – in fact, they are commands. The sole difference is that event handlers cannot be parameterized and so cannot make use of the $ARGn$ macros we've used with previous command definitions.

Event handlers are designed to execute a command in response to a change in service or host state. As such, they can be associated with a host or service using the "event handler" directive:

```
define service{
...
     event_handler    some-command-name
```

or

```
define host{
...
     event-handler      some-command-name
```

Here is an example Event handler:

```
define command {
     command_name     restart-apache
     command_line     /bin/systemctl restart httpd.service
}
```

Note that event handlers are fired for every state change. To invoke an event handler only for specific states, a wrapper script (such as a shellscript) must be used to evaluate the state and call the event handler when appropriate.

Escalations

An escalation refers to the process of notifying an additional group or additional groups of contacts if a problem persists beyond a certain length of time. If you are familiar with technical support, escalation might mean managers or the next tier of of the support or engineering team being alerted.

Escalations provide a way to focus an organization's efforts and ensure that problems don't "slip through the cracks" when encountered.

Nagios Core sends out notifications on a measured time interval, the default specified by the "notification interval" directive in the service or service template. Nagios does not measure response time, but rather the number of notifications sent.

Service Escalations

The following is an example of a service escalation:

```
define serviceescalation{
    host_name               localhost
    service_description     Current Load
    first_notification      5
    last_notification       10
    notification_interval   15
    escalation_options      u,c,r
    contact_group           admins
}
```

And another one:

```
define serviceescalation{
    host_name               localhost
    service_description     Swap Usage
    first_notification      5
    last_notification       10
    notification_interval   15
    escalation_options      u,c,r
    contact_group           admins
}
```

Note that in the case of overlap (where service escalation configurations refer to the same service), Nagios Core will select the smaller interval and notification times. It is important to note that Nagios will respect the time specified by the "notification interval" directive and will not send out escalations if notification sending is disallowed by the configuration.

If you choose to continue escalation notifications until a problem is resolved, be sure to set the "last notification" directive to "0". Be sure to look up these directives in the "Service Escalation Definition

Directives" PDF provided in the course downloads.

Host Escalations

Host escalations are identical to service definitions but apply to an entire host. We create our first host escalation in the exercise "Create a Host Escalation."

Dependencies

Nagios Core provides typically flexible means to define the relationships between and among hosts and services. In particular, this afford Nagios the ability to send out notices intelligently, as we'll see.

There are two categories of relationships in Nagios Core:

- Parent/child: A host may specify its parents; this is used to define infrastructure. For example, a web server may define a load balancer as its parent. Parents may only be defined by hosts.
- Master/dependent: A dependent in this case is a host or service which relies on the master to function properly. Master/dependent relationships can be specified for services and hosts.

Notifications and Dependencies

Nagios can be configured to recognize when parent hosts or master hosts/services fail. In this case, Nagios will generates notifications only for the parents or master host/service, not children and/or dependents. This keeps engineering teams from being deluged by notifications of problems of which they are already aware or which they anticipate.

In terms of the master dependent relationship, services which are dependent can be configured to remain "silent" if a master goes into an error state, regardless of whether or not the master is a host or a service.

Defining Relationships

Be sure to take a look at the "Host Dependency Directives" and "Service Dependency Directives" PDFs in the downloads section of this course.

Host Dependencies (Parent Relationships)

Host dependencies are a unique object type, and some care should be given to where their definition files are placed on the filesystem. That said, let's take a look at our Nagios Core host and examine it for any host dependencies which may exist.

We can identify one right away: the router. Every server we have has the subnet gateway (or router) as a parent. That is, every server we've spun up depends on the router to function. If the router isn't functioning, nothing will. We need to do the following:

- Set up a service to check the router
- Create host dependencies for each of our servers such that the router is a parent to them all

We can do that as follows.

Create a New Host Definition for the Router

We'll need some information before we continue here such as the IPv4 address of the subnet gateway.

We can get this by executing `route -n` on the Nagios Core host, which will then display the routing information:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
0.0.0.0         172.31.96.1     0.0.0.0         UG    0      0        0
eth0
172.31.96.0     0.0.0.0         255.255.240.0   U     0      0        0
eth0
```

Once we have the IP of the router, create a new host definition in the appropriate objects directory (hosts); named it `subnet-gateway-1.cfg`.

```
define host{
    name                            subnet-gateway-1
    host_name                       subnet-gateway-1
    use                             generic-switch
    alias                           subnet-gateway-1
    address                         172.31.96.1
    display_name                    subnet-gateway-1
    contact_groups                  admins
}
```

Note the template used here; if you look at the template definition, you'll see that most of the parameters we need are already defined. As well, the command object specified by the check command directive already exists. This is a great example of how templates can simplify things and save time.

Restart the Nagios service, and check the WebUI to make sure the host has been properly setup.

Create Host Relationships

Now that we've got the subnet gateway set up as host (and is being checked by Nagios Core), we can define the parent relationships for the hosts that rely on it. This will be all of the servers we've spun up so far.

Create a new directory to store host dependencies in the objects directory; name it `host-dependencies`. Make sure the owner and permissions are set correctly. Amend the main configuration file to instruct Nagios to scan this directory for configuration definitions.

Inside that directory, create a new host dependency definition file named `subnet-router-1-children.cfg`. Make sure it looks like this:

```
define hostdependency{
    dependent_hostgroup_name        linux-academy-servers
    host_name                       subnet-gateway-1
    inherits_parent                 1
    execution_failure_criteria      p,o,d,u
    notification_failure_criteria   p,o,d,u
    dependency_period               24x7
}
```

Refer to the "Host Dependency Definition Directives" PDF in the Downloads section of this course for explanations of each directive.

Check the configuration, and restart Nagios. Check the WebUI to see how the changes are reflected; in particular, take a look at the map. (And note that the map layout can be changed in the `cgi.cfg` file.)

Create Service Relationships

Now we're going to create a service dependency for our remote hosts running SSH. We want to indicate that the SSH service is dependent on our router being up and running; if the router goes down, we don't want Nagios to generate notifications or run checks on these services.

Create a new objects directory named `service-dependencies`; make sure it's owned properly and the appropriate permissions are set.

Be sure to add the directory to the main Nagios configuration file using the `cfg_dir` directive.

In the `service-dependencies` directory, create the file `remote-hosts-ssh.cfg`. Edit the file so it has the following contents:

```
define servicedependency{
    dependent_hostgroup_name    remote-hosts
    dependent_service_description   SSH
    host_name           subnet-gateway-1
    service_description             PING
    inherits_parent         1
    execution_failure_criteria  w,c,u
    notification_failure_criteria   w,c,u
    dependency_period       24x7
}
```

Verify your configuration, and restart Nagios. Take a look at the map in the WebUI, as well as the dependency definitions, to see how these changes are reflected.

The NCP exam can be taken online, using a remote proctor. Check the following URL for more information: https://www.nagios.com/services/certification/

Keep in mind that you ARE allowed to use a textbook during this exam. I strongly recommend you printout, staple, and use the Study Guide for this course if you sit for the exam.