



Quick answers to common problems

# Nagios Core Administration Cookbook

## *Second Edition*

Over 90 hands-on recipes that will employ Nagios Core as the anchor of monitoring on your network

Tom Ryder

[PACKT] open source\*  
PUBLISHING community experience distilled



# **Nagios Core Administration Cookbook**

## **Second Edition**

---

# Table of Contents

[Nagios Core Administration Cookbook Second Edition](#)

[Credits](#)

[About the Author](#)

[About the Reviewer](#)

[www.PacktPub.com](#)

[eBooks, discount offers, and more](#)

[Why Subscribe?](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Sections](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Understanding Hosts, Services, and Contacts](#)

[Introduction](#)

[Creating a new network host](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating a new HTTP service](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating a new e-mail contact](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Verifying configuration](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating a new hostgroup](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating a new servicegroup](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Creating a new contactgroup](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Creating a new time period](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Running a service on all hosts on a group](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [2. Working with Commands and Plugins](#)

### [Introduction](#)

#### [Finding a plugin](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

#### [Installing a plugin](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Removing a plugin](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating a new command](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Customizing an existing command](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using an alternative check command for hosts](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Writing a new plugin from scratch](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Implementing threshold checks in a plugin](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using macros as environment variables in a plugin](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

### [3. Working with Checks and States](#)

[Introduction](#)

[Specifying how frequently to check a host or service](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Changing thresholds for PING RTT and packet loss](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Changing thresholds for disk usage](#)

[Getting ready](#)

[How to do it...](#)



[How it works...](#)

[There's more...](#)

[See also](#)

[Scheduling downtime for a host or service](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Managing brief outages with flapping](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Adjusting flapping percentage thresholds for a service](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

#### [4. Configuring Notifications](#)

[Introduction](#)

[Configuring notification periods](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Configuring notifications for groups](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Choosing states for notification](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Specifying the number of failed checks before notification](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Automating contact rotation](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Defining an escalation for repeated notifications](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Defining a custom notification method](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Filtering notifications based on a host or service value](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [5. Monitoring Methods](#)

[Introduction](#)

[Monitoring PING for any host](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Monitoring SSH for any host](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Checking an alternative SSH port](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Monitoring mail services](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Monitoring web services](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Checking that a website returns a given string](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Monitoring database services](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Monitoring the output of an SNMP query](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Monitoring a RAID or other hardware device](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating an SNMP OID for monitoring](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [6. Enabling Remote Execution](#)

[Introduction](#)

[Monitoring local services on a remote machine with NRPE](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Setting the listening address for NRPE](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Setting allowed client hosts for NRPE](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Creating new NRPE command definitions securely](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Giving limited sudo\(8\) privileges to NRPE](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using check\\_by\\_ssh with key authentication instead of NRPE](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using check\\_mk instead of NRPE](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [7. Using the Web Interface](#)

[Introduction](#)

[Using the Tactical Overview](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Viewing and interpreting availability reports](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Viewing and interpreting trends](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Viewing and interpreting notification history](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Adding comments on hosts or services in the web interface](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Viewing configuration in the web interface](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Scheduling checks from the web interface](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more](#)

[See also](#)

[Acknowledging a problem via the web interface](#)

[Getting started](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [8. Managing Network Layout](#)

[Introduction](#)

[Creating a network host hierarchy](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using the network map](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Choosing icons for hosts](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Establishing a host dependency](#)

[Getting ready](#)

[How to do it...](#)



[How it works...](#)

[There's more...](#)

[See also](#)

[Establishing a service dependency](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Monitoring individual nodes in a cluster](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using the network map as an overlay](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [9. Managing Configuration](#)

[Introduction](#)

[Grouping configuration files in directories](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Keeping a configuration under version control](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Configuring host roles using groups](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Building groups using regular expressions](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Using inheritance to simplify configuration](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Defining macros in a resource file](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [Using another object's directives in a host or service check](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using custom directives](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Dynamically building host definitions](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [10. Security and Performance](#)

[Introduction](#)

[Using authentication for the Nagios Core web interface](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Using authenticated contacts](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Writing debugging information to the Nagios log file](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Monitoring Nagios performance with nagiosstats](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Setting up a redundant monitoring host](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

## [11. Automating and Extending Nagios Core](#)

[Introduction](#)

[Allowing and submitting passive checks](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Submitting passive checks from a remote host with NSCA](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Submitting passive checks in response to SNMP traps](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Setting up an event handler script](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Tracking host and service states with Nagiosgraph](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Reading status in a MySQL database with NDOUtils](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Reading status from a Unix socket with MK Livestatus](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Writing customized Nagios Core reports](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Getting extra visualizations with NagVis](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Writing custom Nagios Core management scripts](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Index](#)



# **Nagios Core Administration Cookbook**

## **Second Edition**

---





# **Nagios Core Administration Cookbook**

## **Second Edition**

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2013

Second Edition: February 2016

Production reference: 1230215

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-933-2

[www.packtpub.com](http://www.packtpub.com)



# Credits

## **Author**

Tom Ryder

## **Reviewer**

John C. Kennedy

## **Commissioning Editor**

Priya Singh

## **Acquisition Editor**

Ruchita Bhansali

## **Content Development Editor**

Mayur Pawanikar

## **Technical Editor**

Vivek Pala

## **Copy Editor**

Neha Vyas

## **Project Coordinator**

Nidhi Joshi

## **Proofreader**

Safis Editing

## **Indexer**

Mariammal Chettiyar

## **Graphics**

Disha Haria

## **Production Coordinator**

Nilesh Mohite

## **Cover Work**

Nilesh Mohite



# About the Author

**Tom Ryder** is a systems administrator and former web developer from New Zealand. He uses Nagios Core as part of his “day job” as a systems administrator, monitoring the network for a regional Internet service provider. Tom works a great deal with Unix-like systems, being a particular fan of GNU/Linux, and writes about usage of open source, command-line development tools on his blog, Arabesque, at <http://blog.sanctum.geek.nz>.



# About the Reviewer

**John C. Kennedy** has worked with Unix and Linux since 1998. He has worked on Nagios as a monitoring tool for much of the past 5 years.

He has been reviewing and tech editing books in his spare time since 2001 and has about 20 open source-related books under his belt. He believes the best part of reviewing is that he learns something from every book he works on.

John was born in the U.S. and grew up in Northern Virginia. He spent some time in the U.S. Air Force and has lived in Germany and the United Kingdom. He has been married to Michele since 1994 and has two children, Denise and Kieran. He currently lives in Virginia.

I would like to thank my family, including my nephews, Aiden and Mason, and my niece, Harriet, for supporting all the silly things I do and for giving me time to work on this.

Thanks also to Nidhi Joshi whose great patience and flexibility around my very hectic schedule were VERY much appreciated. Also, my thanks go to the author, Tom Ryder. I learned a lot from the book, and it has been one of my favorite books to work on.

Everyone at Packt Publishing also deserves recognition for all their titles and the hard work that goes into producing every one of them.





**[www.PacktPub.com](http://www.PacktPub.com)**

# eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <[customercare@packtpub.com](mailto:customercare@packtpub.com)> for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

# Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser



# Preface

Nagios Core, the open source version of the Nagios monitoring framework, is an industry standard for network monitoring hosted on Unix-like systems, such as GNU/Linux or BSD. It is very often used by network and system administrators to check the connectivity between hosts and to ensure that network services are running as expected.

Where home-grown scripts performing network checks can rapidly become unmaintainable and difficult for newer administrators to customize safely, Nagios Core provides a rigorous and configurable monitoring framework to make checks in a consistent manner and to alert appropriate people and systems of any problem it detects.

This makes Nagios Core a very general monitoring framework rather than an out-of-the-box monitoring solution, which is known to make it a little unfriendly to beginners and something of a “black box”, even to the otherwise experienced administrators. Busy administrators charged with setting up a Nagios Core system will often set it up to send PING requests to a set of hosts every few minutes and send them an e-mail about any problem, and otherwise never touch it. More adventurous administrators new to the system might instate a few HTTP checks to make sure that company websites respond.

Nagios Core is capable of a great deal more than that, and this book’s recipes are intended to highlight all of the different means of refining and controlling checks, notifications, and reporting for Nagios Core, rather than being a list of instructions for using specific plugins, of which there are many hundreds available online at the Nagios Exchange at <https://exchange.nagios.org/>. The book’s fundamental aim is to get administrators excited about the possibilities of Nagios Core beyond elementary default checking behavior so that they can use much more of the framework’s power and make it into the centerpiece of their network monitoring.

This also includes installing and even writing custom plugins beyond the standard Nagios Plugins set, writing and refining one’s own checks, working with the very powerful **Simple Network Management Protocol (SNMP)**, and the recording and reporting of performance data. It also includes refining notification behavior to only send appropriate notifications at appropriate times to appropriate people or systems, basic visualization options, identifying breakages in network paths, clever uses of the default web interface, and even extending Nagios Core with other open source programs. It includes all this in order to virtually check any kind of host property or network service on any network.

Where possible, this book focuses on add-ons written by the Nagios team themselves, particularly NRPE and NSCA. It omits discussion of popular forks of Nagios Core, such as Icinga. In the interest of conferring an in-depth understanding of advanced Nagios Core configuration, it also does not discuss any configuration frontends or wizards, such as NConf. Finally, as a Packt open source series book focusing on the use of the freely available Nagios Core, it also does not directly discuss the use of Nagios XI, the commercial version of the software supported by the Nagios team. This is done to instill a thorough understanding of Nagios Core itself, rather than to reflect the personal opinions of the author; curious administrators should definitely investigate all of these projects.

# What this book covers

[Chapter 1](#), *Understanding Hosts, Services, and Contacts*, gives you a basic idea of hosts, services, and contacts followed by the summary of this chapter.

[Chapter 2](#), *Working with Commands and Plugins*, explains the architecture of plugins and commands, including installing new plugins and defining custom uses of existing ones. It also walks us through how to write a new plugin with Perl.

[Chapter 3](#), *Working with Checks and States*, explains how Nagios Core performs its checks and how to customize that behavior, including scheduling downtime for hosts and services and managing “flapping” for hosts or services that keep going up and down.

[Chapter 4](#), *Configuring Notifications*, explains the logic of how Nagios Core decides on what basis to notify, when, and to whom, including examples of implementing a custom notification method, escalating notifications that aren’t fixed after a certain period of time, and scheduling contact rotation.

[Chapter 5](#), *Monitoring Methods*, gives examples of the usage of some of the standard Nagios Plugins sets, moving from basic network connectivity checks with PING and HTTP to more complex and powerful checks involving SNMP usage.

[Chapter 6](#), *Enabling Remote Execution*, shows how to use NRPE as a means of working around the problem of not being able to check system properties directly over the network, including a demonstration of the more advanced methods of `check_by_ssh` and `check_mk`.

[Chapter 7](#), *Using the Web Interface*, shows some less-used features of the web interface to actually control how Nagios Core is behaving and to see advanced reports, rather than simply viewing the current state information. Use of the network map is not discussed here but in the next chapter.

[Chapter 8](#), *Managing Network Layout*, explains how to make Nagios Core aware of the structure and function of your network with a focus on hosts and services depending on one another to function correctly, including monitoring clusters and using that layout information to build a network status map, optionally with icons and a background.

[Chapter 9](#), *Managing Configuration*, shows how to streamline, refine, and control Nagios Core’s configuration at a low level without the use of frontends. It focuses on the clever use of groups, templates, macros, and custom directives, and gives an example of generating configuration programmatically with the templating language `m4`.

[Chapter 10](#), *Security and Performance*, shows how to manage simple access control, debug runtime problems, and keep tabs on how Nagios Core is performing, and also contains a demonstration of basic monitoring redundancy.

[Chapter 11](#), *Automating and Extending Nagios Core*, explains how to submit check results from other programs (including NSCA) to provide information about external processes via the commands file, and an introduction to a few popular add-ons (NDOUtils, MK Livestatus, NagVis, and Nagiosgraph).





# What you need for this book

In an attempt to work with a “standard” installation of Nagios Core, this book’s recipes assume that Nagios Core 4.0 or later and the Nagios Plugins set have been installed in `/usr/local/nagios` by the Nagios Quickstart Guides available at

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/quickstart.html>.

If your system’s package repositories include a package for Nagios Core 4.0 or later that you would prefer to use, this should still be possible, but the paths of all the files are likely to be very different. This is known to be a particular issue with the `nagios3` package on Debian or Ubuntu systems. If you are familiar with the differences in the installation layout that your packaging system imposes, then you should still be able to follow the recipes with only a few path changes.



# Who this book is for

This book is aimed at system and network administrators who are comfortable with basic Unix-like system administration via the command line. It is best suited for GNU/Linux administrators, but should work fine for BSD administrators too. It has particular focus on the kind of administrator identified in the preface: one who is comfortable working with their Unix-like system, may well have a basic Nagios Core installation ready with some PING checks, and now wants to learn how to use more of the framework's power and understand its configuration in more depth.

Administrators should be comfortable with installing library dependencies for the extensions, plugins, and add-ons discussed in the book. An effort is made to mention any dependencies; however, how these are best installed will depend on the system and its package repository. In almost all cases, this should amount to installing some common libraries and their headers from a packaging system. Debian and Ubuntu package names are given for some more complex cases.

The easier recipes in the first five chapters involve some recap of the basics of configuring Nagios Core objects. Users completely new to Nagios Core who have just installed it will almost certainly want to start with [Chapter 1](#), *Understanding Hosts, Services, and Contacts*, after completing the Nagios Quickstart Guide, as the latter chapters assume a fair amount of knowledge.



# Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

# Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

## How to do it...

This section contains the steps required to follow the recipe.

## How it works...

This section usually consists of a detailed explanation of what happened in the previous section.



## **There's more...**

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

## See also

This section provides helpful links to other useful information for the recipe.



# Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows: “Nagios Core will only need whatever information the `ping(8)` tool would need for its own `check_ping` command.”

A block of code is set as follows:

```
define service {  
    use                generic-service  
    host_name          sparta.example.net  
    service_description HTTP  
    check_command      check_http  
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
define host {  
    host_name          sparta.example.net  
    alias              sparta  
    address            192.0.2.21  
    max_check_attempts 3  
    check_period       24x7  
    check_command      check-host-alive  
    contacts           nagiosadmin  
    notification_interval 60  
    notification_period 24x7  
}
```

Any command-line input or output is written as follows:

```
# cd /usr/local/nagios/etc/objects  
# vi sparta.example.net.cfg
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: “If the server restarted successfully, the web interface should show a brand new host in the **Hosts** list, in the **PENDING** state as it waits to run a check that the host is alive.”

## Note

Warnings or important notes appear in a box like this.

## Tip

Tips and tricks appear like this.



# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <[feedback@packtpub.com](mailto:feedback@packtpub.com)>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at [www.packtpub.com/authors](http://www.packtpub.com/authors).



# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.



## Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

# Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

# Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <[copyright@packtpub.com](mailto:copyright@packtpub.com)> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

# Questions

If you have a problem with any aspect of this book, you can contact us at [<questions@packtpub.com>](mailto:questions@packtpub.com), and we will do our best to address the problem.



# Chapter 1. Understanding Hosts, Services, and Contacts

In this chapter, we will cover the following recipes:

- Creating a new network host
- Creating a new HTTP service
- Creating a new e-mail contact
- Verifying configuration
- Creating a new hostgroup
- Creating a new servicegroup
- Creating a new contactgroup
- Creating a new time period
- Running a service on all hosts on a group

# Introduction

Nagios Core is appropriate for monitoring services and states on all sorts of hosts and one of its primary advantages is that the configuration can be as simple or as complex as required. Many Nagios Core users will only ever use the software as a way to send PING requests to a few hosts on their local network, or possibly the Internet, and use it to send an administrator an e-mail or pager message if they don't get any replies. Nagios Core is vastly capable of monitoring more complex systems than this, scaling from simple LAN configurations to being the cornerstone for monitoring for an entire network operations team.

However, for both simple and complex configurations of Nagios Core, the most basic building blocks of configuration are **hosts**, **services**, and **contacts**. These are the three things that administrators of even very simple networking setups will end up editing and probably creating. If you're a beginner to Nagios Core, you might have changed a hostname here and there or copied a stanza in a configuration to get it to do what you want; in this first chapter, we're going to look at what these configurations do in a bit more depth than that.

In a Nagios Core configuration, hosts, services, and contacts are three fundamental types of objects:

- **Hosts** usually correspond to a computer, whether it's the machine running Nagios Core itself, a virtual machine hosted upon it, or a machine reachable via a network. Conceptually, however, a host can monitor any kind of network entity, such as the endpoint of a VPN.
- **Services** usually correspond to an arrangement for Nagios Core to check something about a host; this can either be something as simple as whether the monitoring server can get PING replies from the host or something more complicated as whether the value of an SNMP OID is within acceptable bounds.
- **Contacts** define a means to notify someone when events take place on the services on our hosts, such as not being able to get a PING response or send a test e-mail message.

In this chapter, we'll add all three of these configurations, we'll learn how to group their definitions together to make the configuration more readable, and to work with hosts in groups rather than having to edit each one individually. We'll also set up a custom time period for notifications so that hardworking system administrators like us don't end up getting paged at midnight unnecessarily!





# Creating a new network host

In this recipe, we'll start with the default Nagios Core configuration and set up a host definition for a server that responds to PING on our local network. The end result will be that Nagios Core will add our new host to its internal tables when it starts up and will automatically check it (probably using PING) on a regular basis. In this example, I'll use the example of my Nagios Core monitoring server with the DNS name `olympus.example.net` and add a host definition for a web server with the DNS name `sparta.example.net`. This is all on an example network `192.0.2.0/24`.

# Getting ready

You'll need a working Nagios Core 4.0 or greater installation with a web interface and all the Nagios Core plugins installed. If you have not yet installed Nagios Core, you should start with the quick start guide at

<http://nagios.sourceforge.net/docs/nagioscore/4/en/quickstart.html> that is appropriate to your operating system.

We'll assume that the configuration file Nagios Core reads on startup is at `/usr/local/nagios/etc/nagios.cfg`, as is the case with the default installation. It shouldn't matter where you include this new host definition in the configuration, as long as Nagios Core is going to read the file at some point. However, it might be a good idea to give each host its own file in a separate objects directory, which we'll do here. You should have access to a shell on the server and be able to write text files using an editor of your choice; I'll use `vi`. You will need root privileges on the server via `su` or `sudo`.

You should know how to reload Nagios Core on the server so that the configuration you're going to add gets applied. It shouldn't be necessary to restart the whole machine to do this! A common location for the startup/shutdown script on Unix-like hosts is `/etc/init.d/nagios`, which I'll use here. On modern GNU/Linux systems, it may be a better practice to use `systemd` `nagios reload`.

You should also get the hostname or IP address of the server you'd like to monitor ready. We'll use IP addresses rather than DNS hostnames here, which means that our checks will keep working even if DNS is unavailable. You may prefer to use hostnames if your addresses change regularly. You shouldn't need the subnet mask or anything like that; Nagios Core will only need whatever information the `ping(8)` tool would need for its own `check_ping` command.

Finally, you should test things first; confirm that you're able to reach the host from the Nagios Core server using `ping(8)` by checking directly from the shell, to make sure your network stack, routes, firewalls, and netmasks are all correct:

```
user@olympus:~$ ping 192.0.2.21
PING sparta.example.net (192.0.2.21) 56(84) bytes of data.
64 bytes from sparta.example.net (192.0.2.21): icmp_req=1 ttl=64 time=0.149
ms
```

# How to do it...

We can create the new host definition for `sparta.example.net` as follows:

1. Change the directory to `/usr/local/nagios/etc/objects` and create a new file called `sparta.example.net.cfg`:

```
# cd /usr/local/nagios/etc/objects
# vi sparta.example.net.cfg
```

2. Write the following code into the file, changing the values in bold as appropriate for your own setup:

```
define host {
    host_name      sparta.example.net
    alias          sparta
    address        192.0.2.21
    max_check_attempts 3
    check_period   24x7
    check_command  check-host-alive
    contacts      nagiosadmin
    notification_interval 60
    notification_period 24x7
}
```

3. Change the directory to `/usr/local/nagios/etc` and edit the `nagios.cfg` file:

```
# cd ..
# vi nagios.cfg
```

At the end of the file, add the following line:

```
cfg_file=/usr/local/nagios/etc/objects/sparta.example.net.cfg
```

4. Reload the configuration:

```
# /etc/init.d/nagios reload
```

If the server restarted successfully, the web interface should now show a brand new host in the hosts list and a PENDING state as it waits to verify that the host is alive:

| Host                | Status  | Last Check  | Duration  | Status Information                                    |
|--------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|-------------------------------------------------------|
| localhost           | UP                                                                                         | 06-01-2015 17:22:27                                                                            | 0d 0h 0m 21s+                                                                                | PING OK - Packet loss = 0%, RTA = 0.04 ms             |
| sparta.example.net  | PENDING                                                                                    | N/A                                                                                            | 0d 0h 0m 21s+                                                                                | Host check scheduled for Mon Jun 1 17:24:56 NZST 2015 |

In the next few minutes, the host's background should change to green to show that the verification was complete and the host status should change to UP, assuming that the checks succeeded:

| Host               | Status | Last Check          | Duration      | Status Information                        |
|--------------------|--------|---------------------|---------------|-------------------------------------------|
| localhost          | UP     | 06-01-2015 17:22:44 | 0d 0h 2m 3s   | PING OK - Packet loss = 0%, RTA = 0.05 ms |
| sparta.example.net | UP     | 06-01-2015 17:24:58 | 0d 0h 3m 18s+ | PING OK - Packet loss = 0%, RTA = 0.26 ms |

If the test failed and Nagios Core was not able to get a PING response from the target machine after three tries, for whatever reason, it would probably look something like this:

| Host               | Status | Last Check          | Duration     | Status Information                          |
|--------------------|--------|---------------------|--------------|---------------------------------------------|
| localhost          | UP     | 06-01-2015 17:27:48 | 0d 0h 4m 15s | PING OK - Packet loss = 0%, RTA = 0.04 ms   |
| sparta.example.net | DOWN   | 06-01-2015 17:27:47 | 0d 0h 0m 9s  | CRITICAL - Network Unreachable (192.0.2.21) |

# How it works...

The configuration we included in the preceding adds a host to Nagios Core's list of hosts to check, Nagios Core will periodically send a PING request to 192.0.2.21, checking whether it receives a reply, and will update the status as shown in the Nagios Core web interface appropriately. We have neither defined any other services to check for this host yet, nor have we specified what action it should take if the host is down. However, the host itself will be automatically checked at regular intervals by Nagios Core and we can view its state in the web interface at any time.

The directives we defined in the preceding configuration are as follows:

- **host\_name:** This defines the hostname of the machine that is used internally by Nagios Core to refer to this host. It will end up being used in other parts of the configuration.
- **alias:** This defines a more recognizable human-readable name for the host; this appears in the web interface. It could also be used for a full-text description of the host.
- **address:** This defines the IP address of the machine. This is the actual value that Nagios Core will use to contact the server; using an IP address rather than a DNS name is generally a best practice, so the checks continue to work even if DNS is not functioning. In Nagios 4.0 or newer, if you leave this field blank, the value of **host\_name** will be used instead. Before using Nagios 4.0, you must define it.
- **max\_check\_attempts:** This defines the number of times Nagios Core should try to run the check if the checks fail. Here, we've defined a value of 3, meaning that Nagios Core will make a total of three attempts to contact the host before flagging it as DOWN.
- **check\_period:** This references the time period during which this host should be checked. The 24x7 time period is defined in the default configuration for Nagios Core. This is a sensible value for hosts, as it means the host will always be checked. This defines how often Nagios Core will check the host, not how often it will notify anyone.
- **check\_command:** This references the command that will be used to check whether the host is UP, DOWN, or UNREACHABLE. In this case, a standard Nagios Core configuration defines `check-host-alive` as a PING check, which suits as a good test of basic network connectivity and a sensible default for most hosts. This directive is actually not required to make a valid host, but you will want to include it under most circumstances; without it, no checks will be run.
- **contacts:** This references the contact or contacts that will be told about state changes in the host. In this instance, we've used `nagiosadmin`, which is defined in the default Nagios Core configuration.
- **notification\_interval:** This defines how regularly the host should repeat its notifications if it is having problems. Here, we've used a value of 60, which corresponds to 60 minutes, or 1 hour.
- **notification\_period:** This references the time period during which Nagios Core

should send out notifications if there are problems. Here, we're again use the 24x7 time period, but for other hosts, another time period such as `workhours` might be more appropriate.

Note that we added the definition in its own file called `sparta.example.net.cfg` and then referred to it in the main configuration file `nagios.cfg`. This is simply a conventional way of laying out hosts and it happens to be a tidy way to manage things to keep definitions in their own files.

# There's more...

There are a lot of other useful parameters for hosts, but the ones we've used include everything that's required.

While this is a perfectly valid way of specifying a host, it's more typical to define a host based on a **template**, with definitions of how often the host should be checked, who should be contacted when its state changes and on what basis, and similar properties. Nagios Core defines a simple template host called `generic-host`, which could be used by extending the host definition, as with the `use` directive:

```
define host {  
    use                generic-host  
    name               sparta  
    host_name          sparta.example.net  
    address            192.0.2.21  
    max_check_attempts 3  
    contacts           nagiosadmin  
    check_period        24x7  
    check_command       check-host-alive  
}
```

This uses all the parameters defined for `generic-host` and then adds on the details of the specific host that needs to be checked. If you're curious to see what's defined in `generic-host`, you'll find its definition by navigating to `/usr/local/nagios/etc/objects/templates.cfg`.

## See also

- *Specifying how frequently to check a host, [Chapter 3](#), Working with Checks and States*
- *Using an alternative check command for hosts, [Chapter 2](#), Working with Commands and Plugins*
- *Grouping configuration files in directories, [Chapter 9](#), Managing Configuration*
- *Using inheritance to simplify configuration, [Chapter 9](#), Managing Configuration*





# Creating a new HTTP service

In this recipe, we'll create a new service to check on an existing host. Specifically, we'll check our `sparta.example.net` server to verify that it's responding to HTTP requests on the usual HTTP TCP port of 80. To do this, we'll be using a predefined command called `check_http`, which in turn uses one of the standard set of Nagios Core plugins, also called `check_http`. If you don't yet have a web server defined as a host in Nagios Core, you may like to try the recipe *Creating a new network host* in this chapter first.

After we've done this, not only will our host be checked for a PING response with its `check_command`, but Nagios Core will also run a periodic check to ensure that a HTTP service on that machine is responding to requests on the same host.

# Getting ready

You'll need a working Nagios Core 4.0 or greater installation with a web interface, all the Nagios Plugins installed, and at least one host defined. If you need to set up a host definition for your web server first, you might like to read the *Creating a new network host* recipe in this chapter, for which the requirements are the same.

It would be a good idea to test that the Nagios Core server is actually able to contact the web server first, so we know that the test we're about to set up should succeed. The `telnet(1)` tool is a fine way to test that a response comes back from the TCP port 80 as we would expect from a web server:

```
user@olympus:~$ telnet 192.0.2.21 80
Trying 192.0.2.21...
Connected to 192.0.2.21.
Escape character is '^['.
```

# How to do it...

We can create the service definition for `sparta.example.net` as follows:

1. Change to the directory containing the file in which the `sparta.example.net` host is defined and edit it:

```
# cd /usr/local/nagios/etc/objects
# vi sparta.example.net.cfg
```

2. Add the following to the end of the file, substituting the value of the host's `host_name` directive:

```
define service {
    host_name          sparta.example.net
    service_description HTTP
    check_command       check_http
    max_check_attempts 3
    check_interval      5
    retry_interval      1
    check_period        24x7
    notification_interval 60
    notification_period 24x7
    contacts            nagiosadmin
}
```

3. Reload the configuration:

```
# /etc/init.d/nagios reload
```

If the server restarted successfully, the web interface should now show you a new service under the **Services** section and a PENDING state as the service awaits its first check:

|                    |      |         |     |              |     |                                                          |
|--------------------|------|---------|-----|--------------|-----|----------------------------------------------------------|
| sparta.example.net | HTTP | PENDING | N/A | 0d 0h 0m 9s+ | 1/3 | Service check scheduled for Mon Jun 1 17:46:12 NZST 2015 |
|--------------------|------|---------|-----|--------------|-----|----------------------------------------------------------|

Within a few minutes, the service's state should change to OK once the check has run and succeeded with an `HTTP/1.1 200 OK` response, or a similar response:

|                    |      |    |                     |             |     |                                                                      |
|--------------------|------|----|---------------------|-------------|-----|----------------------------------------------------------------------|
| sparta.example.net | HTTP | OK | 06-01-2015 17:44:18 | 0d 0h 0m 9s | 1/3 | HTTP OK: HTTP/1.1 200 OK - 11378 bytes in 0.001 second response time |
|--------------------|------|----|---------------------|-------------|-----|----------------------------------------------------------------------|

If the check had problems, perhaps because the HTTP daemon isn't running on the target server, the check may show **CRITICAL** instead. This probably doesn't mean that the configuration is broken; it more likely means that the network or web server isn't working:

|                    |      |          |                     |              |     |                                                                     |
|--------------------|------|----------|---------------------|--------------|-----|---------------------------------------------------------------------|
| sparta.example.net | HTTP | CRITICAL | 06-01-2015 17:45:42 | 0d 0h 0m 10s | 1/3 | connect to address<br>192.0.2.21 and port 80:<br>Connection refused |
|--------------------|------|----------|---------------------|--------------|-----|---------------------------------------------------------------------|

# How it works...

The configuration we've added adds a simple service check definition for an existing host, to check up to three times whether the HTTP daemon on that host is responding to a simple HTTP/1.1 request. If Nagios Core can't get a response to its test, it will flag the state of the service as `CRITICAL` and will try again two more times before sending a notification. The service will be visible in the Nagios Core web interface, we can check its state any time, and Nagios Core will continue testing the server on a regular basis and flagging whether the checks were successful or not.

It's important to note that the service is like a property of a particular host; we define a service to check for a specific host; in this case, the `sparta.example.net` web server. That's why it's important to get the definition for `host_name` right.

The directives we defined in the preceding configuration are as follows:

- `host_name`: This references the host definition for which this service should apply. This will be the same as the `host_name` directive for the appropriate host.
- `service_description`: This is the name of the service itself, something human-recognizable that will appear in alerts and in the web interface for the service. In this case, we've used `HTTP`.
- `check_command`: This references the command that should be used to check the service's state. Here, we're referring to a command defined in Nagios Core's default configuration called `check_http`, which refers to a plugin of the same name in the Nagios Core Plugins set.
- `max_check_attempts`: This defines the number of times Nagios Core should attempt to recheck the service after finding it in a problematic state.
- `check_interval`: This defines how long Nagios Core should wait between checks when the service is OK or after the number of checks given in `max_check_attempts` has been exceeded.
- `retry_interval`: This defines how long Nagios Core should wait between retrying checks after finding them in a problematic state.
- `check_period`: This references the time period during which Nagios Core should run checks of the service. Here, we've used the sensible 24x7 time period, as defined in Nagios Core's default configuration. Note that this can be different from `notification_period`; we can check the service's status without necessarily notifying a contact.
- `notification_interval`: This defines how long Nagios Core should wait between resending notifications when a service is in a state other than OK.
- `notification_period`: This references the time period during which Nagios Core should send notifications if it finds a host in a state that is not OK. Here, we've again used 24x7, but for some less critical services it might be appropriate to use a time period such as `workhours`.

Note that we added the service definition in the same file as defining the host, directly after it. We can actually place the definition anywhere we like, but this happens to be a

good way to keep things organized.

## There's more...

The service we've set up to monitor on `sparta.example.net` is an HTTP service, but that's just one of the many possible services we could monitor on our network. Nagios Core defines many different commands for its core plugin set, such as `check_smtp`, `check_dns`, and others; all these commands, in turn, point to the programs that actually perform the check and return the results to the Nagios Core server to be dealt with. The important thing to take away from this is that a service can monitor pretty much anything and there are hundreds of plugins available for common network monitoring checks available on the Nagios Exchange website (<http://exchange.nagios.org/>).

There are a great deal more possible directives for services. In practice, we often want to have a service template object with common values, and then extend it for each service we need to check. This allows us to define values that we might want for a number of services, such as how long they should be in a `CRITICAL` state before a notification event takes place and someone gets contacted to deal with the problem.

One such template that Nagios Core's default configuration defines is called `generic-service` and we can use it as a basis for our new service by referring to it with the `use` keyword:

```
define service {  
    use                generic-service  
    host_name          sparta.example.net  
    service_description HTTP  
    check_command      check_http  
}
```

This may work well for you, as there are a lot of very sensible default values set by the `generic-service` template, which makes things a lot easier. We can inspect these values by looking at the template's definition at `/usr/local/nagios/etc/objects/templates.cfg`. This is the same file that includes the `generic-host` definition that we may have used earlier.



## See also

- The *Creating a new servicegroup* section in this chapter
- *Specifying how frequently to check a service*, [Chapter 3](#), *Working with Checks and States*
- *Scheduling downtime for a host or service*, [Chapter 3](#), *Working with Checks and States*
- *Monitoring web services*, [Chapter 5](#), *Monitoring Methods*



# Creating a new e-mail contact

In this recipe, we'll create a new contact with which hosts and services can interact with each other, chiefly to inform the contact when the state of hosts or services changes. We'll use the simplest example of setting up an e-mail contact and configuring an existing host so that this contact receives an e-mail message when Nagios Core's host checks fail and the host is apparently unreachable. In this instance, we'll arrange for `nagios@example.net` to receive an e-mail message whenever the `sparta.example.net` host goes from the `DOWN` state to the `UP` state, or vice-versa.

## Getting ready

You should have a working Nagios Core 4.0 or better server running with a web interface and at least one host to check. If you need to do this first, refer to the *Creating a new network host* recipe in this chapter.

For this particular kind of contact, you'll also need to have a working SMTP daemon running on the monitoring server, such as Exim or Postfix. You should verify that you're able to send messages to the target address and that they're successfully delivered to the host you expect them to be delivered to.

# How to do it...

We can add a simple new contact to the Nagios Core configuration as follows:

1. Change to Nagios Core's configuration directory; ideally, it should contain a file that's devoted to contacts, such as `contacts.cfg` here, and edit that file:

```
# cd /usr/local/nagios/etc/objects
# vi contacts.cfg
```

2. Add the following contact definition to the end of the file, substituting your own values for the properties in bold as you need them:

```
define contact {
    contact_name           spartaadmin
    alias                  Administrator of sparta.example.net
    email                  nagios@example.net
    host_notification_commands    notify-host-by-email
    host_notification_options     d,u,r
    host_notification_period      24x7
    service_notification_commands notify-service-by-email
    service_notification_options  w,u,c,r
    service_notification_period   24x7
}
```

3. Edit the definition for the `sparta.example.net` host and add or replace the definition of contacts for the appropriate host to our new contact `spartaadmin`:

```
define host {
    host_name           sparta.example.net
    alias               sparta
    address              192.0.2.21
    max_check_attempts   3
    check_period         24x7
    check_command         check-host-alive
    contacts            spartaadmin
    notification_interval 60
    notification_period   24x7
}
```

4. Reload the configuration:

```
# /etc/init.d/nagios reload
```

When we are done with the preceding steps, the next time our host changes its state we should receive messages like the one shown in the following screenshot:

```
Date: Mon, 1 Jun 2015 18:19:59 +1200 (NZST)
From: nagios@olympus.example.net
To: nagios@example.net
Subject: ** PROBLEM Host Alert: sparta.example.net is DOWN **

***** Nagios *****

Notification Type: PROBLEM
Host: sparta.example.net
State: DOWN
Address: 192.0.2.21
Info: CRITICAL - Network Unreachable (192.0.2.21)

Date/Time: Mon Jun 1 18:19:59 NZST 2015
```

When the host becomes available again, we should receive a recovery message as follows:

```
Date: Mon, 1 Jun 2015 18:21:53 +1200 (NZST)
From: nagios@olympus.example.net
To: nagios@example.net
Subject: ** RECOVERY Host Alert: sparta.example.net is UP **

***** Nagios *****

Notification Type: RECOVERY
Host: sparta.example.net
State: UP
Address: 192.0.2.21
Info: PING OK - Packet loss = 0%, RTA = 0.27 ms

Date/Time: Mon Jun 1 18:21:53 NZST 2015
```

If possible, it's worth testing this setup with a test host that we can safely bring down and then up again to verify that we receive appropriate notifications.

# How it works...

This configuration adds a new contact to the Nagios Core configuration and references it in one of the hosts as the appropriate contact to be used when the host has problems.

We've defined the required directives for the contact and a couple of others:

- `contact_name`: This defines a unique name for the contact so that we can refer to it in host and service definitions, or anywhere else we might need to do so in the Nagios Core configuration.
- `alias`: This defines a human-friendly name for the contact, perhaps a brief explanation of who the person or group is and/or what they're responsible for.
- `email`: This defines the e-mail address of the contact, since we're going to be sending messages by e-mails.
- `host_notification_commands`: This defines the command or commands to be run when a state change on a host prompts a notification for this contact. In this case, we're going to send e-mails to the the contact about the results with a predefined command called `notify-host-by-email`.
- `host_notification_options`: This specifies different kinds of host events for which this contact should be notified. Here, we're using `d,u,r`, which means that this contact will receive notifications for a host going down, becoming unreachable, or coming back up.
- `host_notification_period`: This defines the time period in which this contact can be notified by any host events. If a host notification is generated and defined to be sent to this contact, but falls outside this time period, the notification will not be sent.
- `service_notification_commands`: This defines the command or commands that are to be run when a state change on a service prompts a notification for this contact. In this case, we're going to send an e-mail to the contact about the results with a predefined command called `notify-service-by-email`.
- `service_notification_options`: This specifies different kinds of service events for which this contact should be notified. Here, we're using `w,u,c,r`, which means that we want to receive notifications about services entering WARNING, UNKNOWN, or CRITICAL states, and also when they recover and go back to the OK state.
- `service_notification_period`: This is the same as for `host_notification_period`, except that this directive refers to notifications about services, not hosts.

Note that we placed the definition for the contact in `contacts.cfg`, which is a reasonably sensible place. However, we can place the contact definition in any file that Nagios Core will read as part of its configuration; we can organize our hosts, services, and contacts any way we like, but it helps to choose some sort of system, so we can easily identify where definitions are likely to be when we need to add, change, or remove them.

## There's more...

If we define a lot of contacts with similar options, it may be appropriate to have individual contacts extend contact templates, so they can inherit those common settings. The default Nagios Core configuration includes such a template, called `generic-contact`. We could instead define our new contact as an extension of this template as follows:

```
define contact {  
    use                generic-contact  
    contact_name       spartaadmin  
    alias              Administrator of sparta.example.net  
    email              nagios@example.net  
}
```

To see the directives defined for `generic-contact`, you can inspect its definition in the `/usr/local/nagios/etc/objects/templates.cfg` file.



## See also

- The *Creating a new contactgroup* section in this chapter
- *Automating contact rotation*, [Chapter 4](#), *Configuring Notifications*
- *Defining an escalation for repeated notifications*, [Chapter 4](#), *Configuring Notifications*



# Verifying configuration

In this recipe, you'll learn the most basic step in debugging a Nagios Core configuration, which is to verify it. This is a very useful step to take before restarting the Nagios Core server to load an altered configuration because it will warn us about possible problems. This is a good recipe to follow if you're not able to start the Nagios Core server at any point because of configuration problems and instead get output like this:

```
# /etc/init.d/nagios reload
```

```
Running configuration check... CONFIG ERROR!  Check your Nagios  
configuration.
```

# Getting ready

You should have a working Nagios Core 4.0 or better server running.

# How to do it...

We can verify the Nagios Core configuration as follows:

1. Run the following command, substituting the path to the nagios binary and our primary nagios.cfg configuration file if necessary:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

2. If the output is very long, it might be a good idea to pipe it through a pager program, such as less:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg | less
```

3. Inspect the output and look for warnings and problems. Here's an example of the part of the output we can expect if our configuration is correct:

```
Checking global event handlers...
Checking obsessive compulsive processor commands...
Checking misc settings...

Total Warnings: 0
Total Errors:   0

Things look okay - No serious problems were detected during the pre-flight check
```

If there's a problem of some sort, we might instead see lines like the following, which is just an example of a possible error; here, my configuration is wrong because I neglected to add a contact\_name directive for a new contact:

```
Reading configuration data...
  Read main config file okay...
Error: Could not complete object definition in file '/usr/local/nagios/etc/objects/contacts.cfg' on line 61.
Have you named all your objects?
  Error processing object config files!

***> One or more problems was encountered while processing the config files...

Check your configuration file(s) to ensure that they contain valid
directives and data definitions.  If you are upgrading from a previous
version of Nagios, you should be aware that some variables/definitions
may have been removed or modified in this version.  Make sure to read
the HTML documentation regarding the config files, as well as the
'Whats New' section to find out what has changed.
```

## How it works...

The configuration is parsed as though Nagios Core were about to start up, to check that the configuration all makes sense. It will run basic checks, such as looking for syntax errors, and will also check things such as having at least one host and service to monitor. Some of the things it reports are warnings, meaning that they're not necessarily problems, such as hosts not having any services monitored or not reporting to any contacts.

This is the quickest way to get an idea of whether the Nagios Core configuration is sane and will work correctly and whenever, there's trouble restarting the Nagios Core server, it's a good idea to check the output of this command.

## There's more...

The program at `/usr/local/nagios/bin/nagios` is actually the same program that runs the Nagios Core server, but the `-v` part of the command is a switch for the program that, instead of starting the server, verifies the configuration and shows any problems with it. The second path is to the configuration file with which Nagios Core starts, which in turn imports configuration files for objects such as the contact, host, and service definitions.

## See also

- The *Writing debugging information to the Nagios Core log file* recipe, [Chapter 10](#), *Security and Performance*





# Creating a new hostgroup

In this recipe, you'll learn how to create a new hostgroup; in this case, to group two web servers together. It is useful to have distinct groups of hosts that might have different properties, such as being monitored by different teams or running different types of monitored services. It also allows us to view a group breakdown in the Nagios Core web interface and to apply a single service to a whole group of hosts rather than doing so individually. This means that all we would have to do to get a new host monitored in the same way as all the other hosts would be to add it to the group, rather than having to specify the configuration manually.

# Getting ready

You should have a working Nagios Core 4.0 or better server running with a web interface.

You should also have at least two hosts that form a meaningful group; perhaps they're similar kinds of servers, such as web servers, or are monitored by the same team, or both at a physical location.

In this example, we have two web servers, `sparta.example.net` and `athens.example.net`, and we're going to add them to a group called `webservers`.

# How to do it...

We can add our new hostgroup webserver to the Nagios Core configuration as follows:

1. Create a new file called `/usr/local/nagios/etc/objects/hostgroups.cfg`, if it doesn't already exist:

```
# cd /usr/local/nagios/etc/objects
# vi hostgroups.cfg
```

2. Write the following code into the new file, substituting the names in bold to suit your own layout:

```
define hostgroup {
    hostgroup_name    webserver
    alias             Webserver with Greek names
}
```

3. Move a directory up and then edit the `nagios.cfg` file:

```
# cd ..
# vi nagios.cfg
```

4. Add this line to the end of the file:

```
cfg_file=/usr/local/nagios/etc/objects/hostgroups.cfg
```

5. For each of the hosts we want to add to the group, find their definitions and add a `hostgroups` directive to put them into the new hostgroup. In this case, our definitions for `sparta.example.net` and `athens.example.net` ends up looking like this:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address             192.0.2.21
    hostgroups        webserver
}
define host {
    use                linux-server
    host_name          athens.example.net
    alias              athens
    address             192.0.2.22
    hostgroups        webserver
}
```

6. Restart Nagios:

```
# /etc/init.d/nagios reload
```

We should now be able to visit the Host Groups section of the web interface and see a new hostgroup with two members:

Webservers with Greek names (webservers)

| Host               | Status | Services             | Actions                                                                                                                                                                                                                                               |
|--------------------|--------|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| athens.example.net | UP     | No matching services |    |
| sparta.example.net | UP     | No matching services |    |

## How it works...

The preceding configuration that we have added includes a new file with a new hostgroup into the Nagios Core configuration and inserts appropriate hosts into the group. The hostgroup creates a separate section in the web interface for us to get a quick overview of only the hosts in that particular group.

## There's more...

The way we've added hosts to the preceding groups is actually not the only way to do it. If we prefer, we can instead name the hosts for the group inside the group definition, using the `members` directive, so we could have something like the following:

```
define hostgroup {
    hostgroup_name    webservers
    alias             Webservers with Greek names
    members           athens.example.net, sparta.example.net
}
```

We can also make a hostgroup that always includes every single host, if we find that useful:

```
define hostgroup {
    hostgroup_name    all
    alias             All hosts
    members           *
}
```

If we're going to be using hostgroups extensively in our Nagios Core configuration to add hosts to groups, we should use whichever of the two methods we think is going to be easiest for us to maintain.

It's worth noting that a host can be in more than one group and there is no limit on the number of groups we can declare, so we can afford to be quite liberal with how we group our hosts into any sort of useful categories. examples of hostgroups could be organizing servers by a function, manufacturer, or colocation customer or organizing routers by BGP or OSPF usage; it all depends on what kind of network we're monitoring.

## See also

- The *Creating a new host* section in this chapter
- The *Running a service on all hosts in a group* section in this chapter
- *Using inheritance to simplify a configuration*, [Chapter 9](#), *Managing Configuration*





# Creating a new servicegroup

In this recipe, we'll create a new servicegroup. This allows us to make meaningful groups out of a set of arbitrary services so that we can view the status of all those services in a separate part of the web administration area.

# Getting ready

You should have a working Nagios Core 4.0 or better server running with a web interface.

You should also have at least two defined services that form a meaningful group; perhaps they're similar kinds of services, such as mail services, are monitored by the same team, or they are both on the same set of servers at a physical location.

In this example, we have three servers performing mail functions: `smtp.example.net`, `pop3.example.net`, and `imap.example.net`, running an SMTP, POP3, and IMAP daemon, respectively. All three of the hosts are set up in Nagios Core, and so are their services. We're going to add them into a new servicegroup called `mailservices`.

Here are the definitions of the hosts and services used in this example, so you can see how everything fits together:

```
define host {
    use                linux-server
    host_name          smtp.example.net
    alias              smtp
    address            192.0.2.31
    hostgroups         webservers
}

define service {
    use                generic-service
    host_name          smtp.example.net
    service_description SMTP
    check_command      check_smtp
}

define host {
    use                linux-server
    host_name          pop3.example.net
    alias              pop3
    address            192.0.2.32
    hostgroups         webservers
}

define service {
    use                generic-service
    host_name          pop3.example.net
    service_description POP3
    check_command      check_pop
}

define host {
    use                linux-server
    host_name          imap.example.net
    alias              imap
    address            192.0.2.33
    hostgroups         webservers
}

define service {
    use                generic-service
    host_name          imap.example.net
    service_description IMAP
    check_command      check_imap
}
```

}

# How to do it...

We can add our new servicegroup with the following steps:

1. Change to our Nagios Core configuration objects directory and edit a new file called `servicegroups.cfg`:

```
# cd /usr/local/nagios/etc/objects
# vi servicegroups.cfg
```

2. Add the following definition to the new file, substituting the values in bold with your own values:

```
define servicegroup {
    servicegroup_name  mailservices
    alias              Mail services
}
```

3. Move a directory up and then edit the `nagios.cfg` file:

```
# cd ..
# vi nagios.cfg
```

4. Add this line to the end of the file:

```
cfg_file=/usr/local/nagios/etc/objects/servicegroups.cfg
```

5. For each of the services we want to add to the group, find their definitions and add a `servicegroups` directive to put them into the new servicegroup. The definitions may end up looking something like this:

```
define service {
    use                generic-service
    host_name          smtp.example.net
    service_description SMTP
    check_command       check_smtp
    servicegroups      mailservices
}
define service {
    use                generic-service
    host_name          pop3.example.net
    service_description POP3
    check_command       check_pop
    servicegroups      mailservices
}
define service {
    use                generic-service
    host_name          imap.example.net
    service_description IMAP
    check_command       check_imap
    servicegroups      mailservices
}
```

6. Restart Nagios using the following command:

```
# /etc/init.d/nagios reload
```

We should now be able to visit the **Service Groups** section of the web interface and see a new servicegroup with three members:

| Mail services (mailservices) |        |          |                                                                                                                                                                                                                                                       |
|------------------------------|--------|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Host                         | Status | Services | Actions                                                                                                                                                                                                                                               |
| imap.example.net             | UP     | 1 OK     |    |
| pop3.example.net             | UP     | 1 OK     |    |
| smtp.example.net             | UP     | 1 OK     |    |

## How it works...

The configuration we've added in the preceding section includes a new file with a new servicegroup into the Nagios Core configuration and inserts appropriate services into the group. The servicegroup creates a separate section in the web interface for us to get a quick overview of only the services in that particular group.

## There's more...

The way we've added services to the preceding groups is not the only way to do it. If we prefer, we can instead name the services (and their applicable hosts) for the group inside the group definition using the `members` directive, so we could have something like the following:

```
define servicegroup {
    servicegroup_name    mailservices
    alias                 Mail services
    members
smtp.example.net,SMTP,pop3.example.net,POP3,imap.example.net,IMAP
}
```

Note that we need to specify both the host that the service is on and the services to be monitored on it. This string needs to be comma-separated. The hostname always comes first, but it can be followed by any number of its services that are to be monitored.

This allows us to make a servicegroup that always includes every single service on every host, if we find that useful:

```
define servicegroup {
    servicegroup_name    all
    alias                 All services
    members               *
}
```

If we're going to use servicegroup definitions extensively in our Nagios Core configuration to add services to groups, we should use whichever of the two methods we think is going to be easiest for us to maintain.

It's worth noting that a service can be in more than one group and there is no limit on the number of groups we can declare, so we can afford to be quite liberal with how we group our services into any sort of useful categories. A few examples of this could be organizing services by the appropriate contact for their notifications, internal functions, or customer-facing functions.



## See also

- The *Creating a new service* section in this chapter
- The *Running a service on all hosts in a group* section in this chapter
- *Using inheritance to simplify a configuration*, [Chapter 9](#), *Managing Configuration*



# Creating a new contactgroup

In this recipe, we'll create a new contactgroup into which we can add our contacts. Like hostgroups and servicegroups, contactgroups mostly amount to convenient shortcuts. In this case, it allows us to define a contactgroup as the recipient of notifications for a host or service definition. This means that we could define an ops group, for example, and then even if people joined or left the group, we wouldn't need to change any definitions for the hosts or services.

# Getting ready

You should have a working Nagios Core 4.0 or better server running.

You should also have at least two contacts that form a meaningful group. In this case, we have two staff members, John Smith and Jane Doe, who are both part of our network operations team. We want them both to be notified for all the appropriate hosts and services, so we'll add them to a group called ops. Here are the definitions with which we're working:

```
define contact {
    use             generic-contact
    contact_name    john
    alias           John Smith
    email           john@example.net
}
define contact {
    use             generic-contact
    contact_name    jane
    alias           Jane Doe
    email           jane@example.net
}
```

# How to do it...

We can create our new ops contactgroup as follows:

1. Change to our Nagios Core configuration objects directory and edit the `contacts.cfg` file:

```
# cd /usr/local/nagios/etc
# vi contacts.cfg
```

2. Add the following definition to the file, substituting in your own values in bold as appropriate:

```
define contactgroup {
    contactgroup_name ops
    alias              Network operators
}
```

3. For each of the contacts that we want to add to the group, find their definitions and add the `contactgroups` directive to them. The definitions will end up looking something like this:

```
define contact {
    use          generic-contact
    contact_name john
    alias        John Smith
    email        john@example.net
    contactgroups ops
}
define contact {
    use          generic-contact
    contact_name jane
    alias        Jane Doe
    email        jane@example.net
    contactgroups ops
}
```

4. Reload the configuration:

```
# /etc/init.d/nagios reload
```

## How it works...

With this group set up, we are now able to use it in the `contactgroups` directive for hosts and services to define which contacts notifications should be sent to. Notifications are sent to all the addresses in the group. This can replace the `contacts` directive, where we name contacts individually.

## There's more...

As an example, consider the following service definition:

```
define service {  
    use                generic-service  
    host_name          sparta.example.net  
    service_description HTTP  
    check_command      check_http  
    contacts           john, jane  
}
```

Instead of having a service definition as the preceding one, we could use this:

```
define service {  
    use                generic-service  
    host_name          sparta.example.net  
    service_description HTTP  
    check_command      check_http  
    contact_groups     ops  
}
```

If John Smith were to leave the operations team, we could simply remove his contact definition and nothing else would require changing; from then on, only Jane would receive the services notification. This method provides a layer of abstraction between contacts and the hosts and services for which they receive notifications.

## See also

- The *Creating a new contact* section in this chapter
- *Automating contact rotation*, [Chapter 4](#), *Configuring Notifications*
- Using inheritance to simplify a configuration, [Chapter 9](#), *Managing Configuration*





# Creating a new time period

In this recipe, we'll add a new time period definition to the Nagios Core configuration so that it allows us to set up monitoring for hosts and services only during weekdays. There's a default configuration defined as `workhours` that would almost suit us, except it doesn't include evenings, so we'll make a new one from scratch. We'll make another one to cover the weekends, too.

# Getting ready

You should have a working Nagios Core 4.0 or better server running. You should also have a clear idea about when the boundaries for the time period you want to define are.

# How to do it...

We can set up our new time period, which we'll call weekdays, as follows:

1. Change to our Nagios Core configuration objects directory and edit the file called `timeperiods.cfg`:

```
# cd /usr/local/nagios/etc/objects
# vi timeperiods.cfg
```

2. Add the following definitions to the end of the file:

```
define timeperiod {
    timeperiod_name weekdays
    alias Weekdays
    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
}
define timeperiod {
    timeperiod_name weekends
    alias Weekends
    saturday 00:00-24:00
    sunday 00:00-24:00
}
```

3. Reload the configuration:

```
# /etc/init.d/nagios reload
```

## How it works...

In our host and service definitions, there are two directives, `check_period` and `notification_period`, which are used to define the times during which a host or service should be checked and the times when notifications about it should be sent. The two examples normally defined in the stock Nagios Core configuration are the 24x7 period and the workhours period, which are defined in the `timeperiods.cfg` file that we just edited and used in several of the examples and templates.

We've just added two more of these time periods that we can now use in our definitions for hosts and services. The first is called `weekdays`, which corresponds to any time during a weekday; the second is called `weekends`, which corresponds to any time that's not a weekday. Note that, in both cases, we specified the dates and times by naming each individual day and the times to which it corresponded.

# There's more...

The definitions for dates are flexible, and can be defined a variety of ways. The following are all valid syntaxes to define days and time periods:

- `june 1 - july 15 00:00-24:00`: This syntax denotes June 1 to July 15, inclusive
- `thursday -1 00:00-24:00`: This syntax denotes the last Thursday of every month
- `day 1 - 10 13:00-21:00`: This syntax denotes the time from 1 p.m. to 9 p.m. on any day from the 1st day of any month to the 10th day of the same month

In a typical practice, it's likely that the standard 24x7 and workhours definitions will be fine for day-to-day monitoring, maybe with a `weekdays` and `weekends` definition added. However, there may come a time when we need a specific host or service monitored on an unusual schedule, particularly if we're debugging a specific problem that only manifests around a certain time, or have a lot of contacts to manage, or a complex on-call roster.

Note that Nagios Core can behave in unusual ways, particularly with uptime reporting, if the time periods for our monitoring of hosts and services don't add up to 24 hours. Ideally, we should check and notify all our hosts and services in some way around the clock, but by dealing with the notifications in different ways depending on the schedule, for example paging the systems administrators about a noncritical system during work hours but just e-mailing them when they're asleep!

## See also

- Automating contact rotation, [Chapter 4](#), *Configuring Notifications*
- Configuring notification periods, [Chapter 4](#), *Configuring Notifications*
- Configuring notification groups, [Chapter 4](#), *Configuring Notifications*





# Running a service on all hosts on a group

In this recipe, we'll create a new service but, instead of applying it to an existing host, we'll apply it to an existing hostgroup; in this case, to a hostgroup called `webserver`s. The steps to do this are very similar to adding a service for just one host; only one directive is different.

# Getting ready

You should have a working Nagios Core 4.0 or better server running with a web interface. You should be familiar with adding services to individual hosts.

You should also have at least one hostgroup defined, with at least one host in it; we'll use a group called `webserver`s with the `sparta.example.net` and `athens.example.net` hosts defined in it.

For reference, here is the hostgroup definition and the definitions for the two hosts in it:

```
define hostgroup {
    hostgroup_name  webserver
    alias           Webserver
}
define host {
    use             linux-server
    host_name       athens.example.net
    alias           athens
    address         192.0.2.22
    hostgroups      webserver
}
define host {
    use             linux-server
    host_name       sparta.example.net
    alias           sparta
    address         192.0.2.21
    hostgroups      webserver
}
```

# How to do it...

We can create the service definition for the webservers group as follows:

1. Change to the directory containing the file in which the webservers hostgroup is defined and edit it:

```
# cd /usr/local/nagios/etc/objects
# vi hostgroups.cfg
```

2. Add the following just after the hostgroup definition. Change the lines in bold to suit your own template and hostgroup names:

```
define service {
    use                generic-service
    hostgroup_name    webservers
    service_description HTTP
    check_command      check_http
}
```

3. Reload the configuration:

```
# /etc/init.d/nagios reload
```

It's important to note that if we were already monitoring these hosts with a per-host service of the same name, we will need to remove the definitions as well; Nagios Core will not start if a service of the same description is already defined on the same host.

## How it works...

Adding a service to a hostgroup works exactly in the same way as adding it to an individual host, except it only requires one definition, which is then individually applied to all the hosts in the group. This means that it's a very good way to keep a Nagios Core configuration tidier. If we have a group with 50 different web servers in it and we need to monitor their HTTP services on the same basis for every one of them, we don't need to create 50 service definitions as well; we can just create one for its hostgroup, which amounts to a smaller and more easily updated configuration.

## There's more...

Like the `host_name` directive for services, `hostgroup_name` can actually have several hostgroups defined, separated by commas. This means that we can apply the same service to not just one group but several of them. For services that we would want to run on several different groups, for example basic PING monitoring, this can amount to a much more flexible configuration.

## See also

- The *Creating a new service* section in this chapter
- The *Creating a new hostgroup* section in this chapter
- *Using inheritance to simplify a configuration*, [Chapter 9](#), *Managing Configuration*



# Chapter 2. Working with Commands and Plugins

In this chapter, we will cover the following recipes:

- Finding a plugin
- Installing a plugin
- Removing a plugin
- Creating a new command
- Customizing an existing command
- Using an alternative check command for a host
- Writing a new plugin from scratch
- Implementing threshold checks in a plugin
- Using macros as environment variables in a plugin



# Introduction

Nagios Core is perhaps best thought of less as a monitoring tool and more as a monitoring framework. Its modular design allows any kind of program that returns appropriate values based on some kind of check as a `check_command` for a host or service, for use to ensure that a host or service is still responsive and functioning correctly. This is where the concepts of **commands** and **plugins** come into play.

For Nagios Core, a **plugin** is any program that can be used to gather information about a host or service. To ensure that a host is responding to PING requests, we'd use a plugin such as `check_ping`, which when run against a hostname or address—whether by Nagios Core or not—returns a status code to whatever called it, based on whether a response was received to the PING request within a certain period of time. This status code and any accompanying message is what Nagios Core uses to establish what state a host or service is in.

Plugins are generally just like any other program on a Unix-like system; they can be run from the command line, are subject to permissions and owner restrictions, can be written in any language, can read variables from their environment, and can take parameters and options to modify how they work. Most importantly, they are entirely separate from Nagios Core itself (even if programmed by the same people) and the way in which they're used by the application can be changed.

To allow additional flexibility in how plugins are used, Nagios Core uses these programs according to the terms of a **command** definition. A command for a specific plugin defines the way in which that plugin is used, including its location in the filesystem, any parameters that should be passed to it, and any other options. In particular, parameters and options often include thresholds for the WARNING and CRITICAL states.

Nagios Core is usually downloaded and installed alongside a set of plugins called Nagios Plugins, which are available at <https://nagios-plugins.org/>, which this book assumes you have installed. These plugins were chosen because, as a set, they cover the most common needs for a monitoring infrastructure quite well. These common needs include checks for common services, such as web services, mail services, DNS services, and others. They also perform more generic checks, such as whether a TCP or UDP port is accessible and open on a server. It's possible that for most, if not all, of our monitoring needs, we won't need any other plugins. However, if we do need other plugins, Nagios Core makes it possible for us to use the existing plugins in novel ways using custom command definitions, to add third-party plugins written by contributors on the Nagios Exchange website, or even to write custom plugins ourselves from scratch in some special cases.



# Finding a plugin

In this recipe, we'll follow a procedure to find a plugin appropriate to a specific monitoring task. We'll start by checking whether an existing plugin is already available to do just what we need. If we can't find one, we'll check whether we can use another more generic plugin to solve the problem. If we still find that nothing suits, we'll visit Nagios Exchange and search for an appropriate plugin.

# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services that are configured already, and you'll need to have a particular service on one of these hosts, which you're not yet sure you need to monitor.

We'll use a simple problem as an example; we have a server named `troy.example.net` that runs an `rsync(1)` process that listens on port 873. We're already monitoring the host's network connectivity via PING, but we'd like to have Nagios Core check whether the `rsync(1)` server is available and listening at all times, in case it crashes while running or doesn't start up when the system is rebooted.

# How to do it...

We can find a new plugin appropriate to any monitoring task as follows:

1. Firstly, since we have the Nagios Core Plugins set installed, we'll check whether any of the plugins available in it can be applied directly to our problem. We'll start by navigating to `/usr/local/nagios/libexec` on our Nagios Core server and getting a directory listing:

```
# cd /usr/local/nagios/libexec
# ls
check_apt      check_ide_smart  check_nntp     check_simap
check_breeze   check_ifoperstatus check_nntps    check_smtp
check_by_ssh   check_ifstatus   check_nt       check_spop
check_cluster  check_imap       check_ntp_time check_tcp
...
```

There's a long list of plugins there, but none of them look like `check_rsync` or even `check_backup`, so it doesn't quite seem like there's a plugin in the core to do exactly what we need.

2. However, there is a plugin called `check_tcp`. A web search for its name pulls up its manual page on the Nagios Plugins website as the first result and a description of what it does:

*"This plugin tests TCP connections with the specified host (or Unix socket)."*

We need to do more than just check the port, so this doesn't quite suit us either.

3. A web search for `check_rsync`, which would be an appropriate name for the plugin, turns up a page on the Nagios Exchange website with a plugin named exactly that. We've found an appropriate plugin.

**check\_rsync**

Submit review | Recommend | Print | Claim

Rating

★★★★☆ 2 votes

Favoured: 0

Owner **nagiosexchange**

Hits 31510

Files:

| File                | Description        |
|---------------------|--------------------|
| <b>check_rsync</b>  | check_rsync v.1.01 |
| <b>check_rsync2</b> | check_rsync v.1.02 |

Checks rsync servers availability, as well as (optionally) individual modules availability. It also supports authentication on modules.

## How it works...

If all we needed to do was verify that `rsync(1)` was listening on port 873, and we didn't really need to monitor any of its actual function, the `check_tcp` plugin might actually suffice. However, in our case, perhaps we might need to find a way to not only verify that a port is open, but also check whether a specific `rsync(1)` module is available.

After reading the description of `check_rsync`, it looks like the plugin has the exact functionality we need, checking that a certain `rsync(1)` module is available on the server. At this point, we could download the plugin and follow its installation instructions.

## There's more...

This recipe is intended to highlight that, in addition to having a capable set of plugins as part of the Nagios Core Plugins set, the documentation available online on the Nagios Core Plugins website at <https://nagios-plugins.org/> and the other plugins available on Nagios Exchange at <https://exchange.nagios.org/> make it relatively straightforward to find an appropriate plugin for the particular monitoring problem we need to solve.

Note that when we download third-party plugins, it's important to verify that we trust the plugin to do what we need it to. Nagios Exchange is a moderated community, but the plugins are provided at our own risk; if we don't understand what a plugin does, we should be wary of installing it or using it on our system without reading its documentation and possibly its reviews.

## See also

- The *Installing a plugin* section in this chapter
- The *Removing a plugin* section in this chapter
- The *Writing a new plugin from scratch* section in this chapter





# Installing a plugin

In this recipe, we'll install a custom plugin that we retrieved from Nagios Exchange on a Nagios Core server so that we can use it in a Nagios Core command and hence check a service with it.

# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already, and you should have found an appropriate plugin that is to be installed to solve a particular monitoring need. Your Nagios Core server should have Internet connectivity to allow you to download the plugin directly from the website.

In this example, we'll use `check_rsync`, which is available on the Web at [https://exchange.nagios.org/directory/Plugins/Network-Protocols/Rsync/check\\_rsync/details](https://exchange.nagios.org/directory/Plugins/Network-Protocols/Rsync/check_rsync/details).

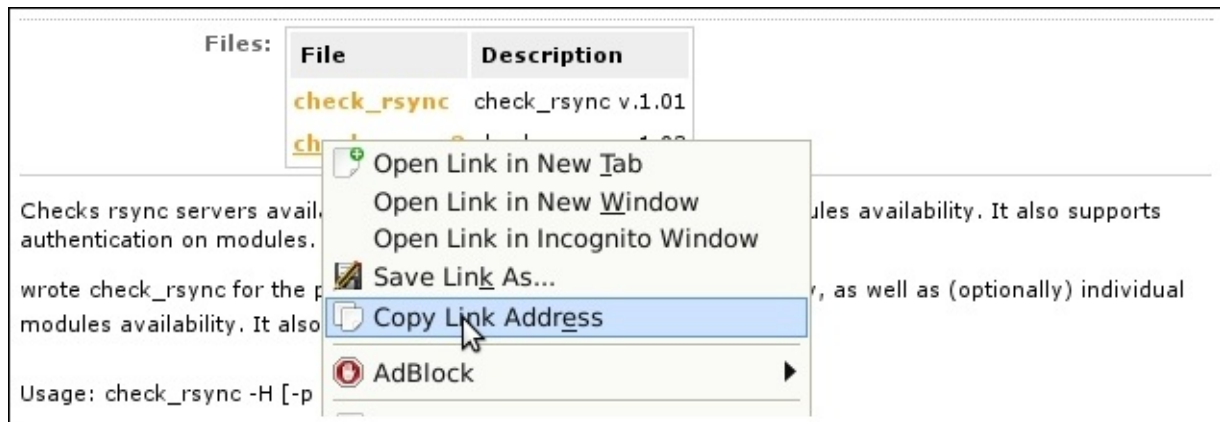
This particular plugin is quite simple, consisting of a single Perl script with only very basic dependencies. If you want to install this script as an example, the server will also need to have a Perl interpreter installed, for example, in `/usr/bin/perl`.

This example will also include directly testing a server running an `rsync(1)` daemon, called `troy.example.net`.

# How to do it...

We can download and install a new plugin as follows:

1. Copy the URL for the download link for the most recent version of the check\_rsync plugin, as shown in the following screenshot:



2. Navigate to the plugins directory for the Nagios Core server. The default location is `/usr/local/nagios/libexec`:

```
# cd /usr/local/nagios/libexec
```

3. Download the plugin using `wget` in a file called `check_rsync`. It's important to surround the URL with single quotes:

```
# wget  
'https://exchange.nagios.org/components/com_mtree/attachment.php?  
link_id=307&cf_id=29' -O check_rsync
```

4. Make the plugin executable using `chmod(1)` and `chown(1)`:

```
# chown root.nagios check_rsync  
# chmod 0770 check_rsync
```

5. Run the plugin directly with no arguments to verify that it runs and to get usage instructions. It's a good idea to test it as the `nagios` user using `su(8)` or `sudo(8)`:

```
# sudo -s -u nagios  
$ ./check_rsync  
Usage: check_rsync -H <host> [-p <port>] [-m <module>[,<user>,  
<password>] [-m <module>[,<user>,<password>]...]]
```

6. Try running the plugin directly against a host running `rsync(1)` to check whether it works and reports a status:

```
$ ./check_rsync -H troy.example.net
```

The output normally starts with the status determined, with any extra information after a colon:

**OK: Rsync is up**

If all of this works, then the plugin is now installed and works correctly.

## How it works...

Because Nagios Core plugins are programs themselves, installing a plugin really amounts to saving a program or script into an appropriate directory; in this case, `/usr/local/nagios/libexec`, where all the other plugins live. It's then available to be used in the same way as any other plugin.

The next step once the plugin is working is defining a command in the Nagios Core configuration for it so that it can be used to monitor hosts and/or services. This can be done with the *Creating a new command* recipe in this chapter.

## There's more...

If we inspect the Perl script, we can see a little bit of how it works. It works like any other Perl script except, perhaps, for the fact that its **return values** are defined in a hash called `%ERRORS`, and the return values it chooses depends on what happens when it tries to check the `rsync(1)` process. This is the most important part of implementing a plugin for Nagios Core.

Installation procedures for different plugins vary. In particular, many plugins are written in languages such as C and hence need be compiled. One such plugin is the popular `check_nrpe` command. Rather than simply being saved into a directory and made executable, these sort of plugins often follow the usual pattern of configuration, compilation, and installation:

```
$ ./configure
$ make
# make install
```

For many plugins that are built in this style, the final step of `make install` will often be to install the compiled plugin in the appropriate directory for us. In general, if instructions are included with the plugin, it pays to read them to see how best to install the plugin.

## See also

- The *Finding a plugin* section in this chapter
- The *Removing a plugin* section in this chapter
- The *Creating a new command* section in this chapter





# Removing a plugin

In this recipe, we'll remove a plugin that we no longer need as part of our Nagios Core installation. Perhaps it's not working correctly, the service it monitors is no longer available, or there are security or licensing concerns with its usage.

# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already, and you should have a plugin that you would like to remove from the server. In this instance, we'll remove the no longer needed `check_rsync` plugin from our Nagios Core server.

# How to do it...

We can remove a plugin from our Nagios Core instance as follows:

1. Remove any part of the configuration that uses the plugin, including hosts or services that use it for a `check_command` and command definitions that refer to the program. As an example, the following definition for a command would no longer work after we removed the `check_rsync` plugin:

```
define command {  
    command_name    check_rsync  
    command_line    $USER1$/check_rsync -H $HOSTADDRESS$  
}
```

Using a tool such as `grep(1)` can be a good way to find mentions of the command and plugin:

```
# grep -R check_rsync /usr/local/nagios/etc
```

2. Change the directory location on the Nagios Core server to wherever the plugins are kept. The default location is `/usr/local/nagios/libexec`:

```
# cd /usr/local/nagios/libexec
```

3. Delete the plugin using `rm(1)`:

```
# rm check_rsync
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

## How it works...

Nagios Core plugins are simply external programs that the server uses to perform checks of hosts and services. If a plugin is no longer wanted, all that we need to do is remove references to it in our configuration, if any, and delete it from `/usr/local/nagios/libexec`.

## **There's more...**

There's not usually any harm in leaving the plugin's program on the server even if Nagios Core isn't using it. It doesn't slow anything down or cause any other problems, and it may be needed later. Nagios Core plugins are generally quite small programs and should not really cause disk space concerns on a modern server.

## See also

- The *Finding a plugin* section in this chapter
- The *Installing a plugin* section in this chapter
- The *Creating a new command* section in this chapter





# Creating a new command

In this recipe, we'll create a new command for a plugin that was just installed into the `/usr/local/nagios/libexec` directory in the Nagios Core server. This will define the way in which Nagios Core should use the plugin and thereby allow it to be used as part of a service definition.

# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already, and you should have a plugin installed for which you'd like to define a new command so that you can use it as part of a service definition. In this instance, we'll define a command for an installed `check_rsync` plugin.

# How to do it...

We can define a new command in our configuration as follows:

1. Change to the directory containing the command configuration files for Nagios Core. The default file is `commands.cfg`, located in `/usr/local/nagios/etc/objects`:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the `commands.cfg` file:

```
# vi commands.cfg
```

3. At the bottom of the file, add the following command definition:

```
define command {  
    command_name    check_rsync  
    command_line    $USER1$/check_rsync -H $HOSTADDRESS$  
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

If the validation passes and the server restarts successfully, we should now be able to use the `check_rsync` command in a service definition.

# How it works...

The configuration we added to the preceding `commands.cfg` file defines a new command called `check_rsync`, which specifies a method for using the plugin of the same name to monitor a service. This enables us to use `check_rsync` as a value for the `check_command` directive in a service declaration, which might look like this:

```
define service {  
    use                generic-service  
    host_name          troy.example.net  
    service_description RSYNC  
    check_command      check_rsync  
}
```

Only two directives are required for command definitions, and we've defined both:

- `command_name`: This defines the unique name with which we can reference the command when we use it in host or service definitions.
- `command_line`: This defines the command line that should be executed by Nagios Core to make the appropriate check.

This particular command line also uses two macros:

- `$USER1$`: This expands to a string value, in our case `/usr/local/nagios/libexec`, the location of the plugin binaries, including `check_rsync`. This is defined in the sample configuration in the `/usr/local/nagios/etc/resource.cfg` file.
- `$HOSTADDRESS$`: This expands to a string value, the address of any host for which this command is used as a host or service definition.

So, if we used the command in a service checking, the `rsync(1)` server on `troy.example.net`, the completed command might look like this:

```
$ /usr/local/nagios/libexec/check_rsync -H troy.example.net
```

We could run this straight from the command line as the `nagios` user to see what kind of results it returns:

```
$ /usr/local/nagios/libexec/check_rsync -H troy.example.net  
OK: Rsync is up
```

## There's more...

A plugin can be used for more than one command. If we had a particular `rsync(1)` module that we wanted to check named `backup`, we could write another command called `check_rsync_backup` as follows, perhaps added to `commands.cfg`:

```
define command {  
    command_name    check_rsync_backup  
    command_line    $USER1$/check_rsync -H $HOSTADDRESS$ -m backup  
}
```

If one or more of our `rsync(1)` servers were running on an alternate port, say port 5873, we could define a separate command `check_rsync_altport` for that:

```
define command {  
    command_name    check_rsync_altport  
    command_line    $USER1$/check_rsync -H $HOSTADDRESS$ -p 5873  
}
```

Commands can thus be defined as precisely as we need them to be. We explore this in more detail in the *Customizing an existing command* recipe in this chapter.

## See also

- The *Installing a plugin* section in this chapter
- The *Customizing an existing command* section in this chapter
- *Creating a new service*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*



# Customizing an existing command

In this recipe, we'll customize an existing command definition. There are a number of reasons why you might want to do this. A common one is if a check is *overzealous*, sending notifications for the WARNING or CRITICAL states which aren't actually terribly worrisome; or, on the other hand, if a check is too *forgiving* and doesn't flag hosts or services as it has problems in recognizing when it would actually be appropriate to do so.

Another reason is to account for peculiarities in your own network. For example, if you run HTTP daemons on a large number of hosts on the alternative port 8080 that you need to check, it would be convenient to have a `check_http_altport` command available. We can do this by copying and altering the definition for the vanilla `check_http` command.



# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already. You should also already be familiar with the relationship between services, commands, and plugins.

# How to do it...

We can customize an existing command definition as follows:

1. Change to the directory containing the command configuration files for Nagios Core. The default file is `commands.cfg`, located in `/usr/local/nagios/etc/objects`:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the `commands.cfg` file or whichever file is an appropriate location for the `check_http` command:

```
# vi commands.cfg
```

3. Find the definition for the `check_http` command. In a default Nagios Core configuration, it should look something like this:

```
# 'check_http' command_definition
define command {
    command_name    check_http
    command_line    $USER1$/check_http -I $HOSTADDRESS$ $ARG1$
}
```

4. Copy this definition into a new definition directly below it and alter it to look like the following, renaming the command and adding a new option to its command line:

```
# 'check_http_altport' command_definition
define command {
    command_name    check_http_altport
    command_line    $USER1$/check_http -I $HOSTADDRESS$ -p 8080 $ARG1$
}
```

5. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

If the validation passes and the server restarts successfully, we should now be able to use the `check_http_altport` command, which is based on the original `check_http` command, in a service definition.

# How it works...

The configuration we added to the preceding `commands.cfg` file reproduces the command definition for `check_http` but changes it in the following two ways:

- It renames the command from `check_http` to `check_http_altport`, which is necessary to distinguish the commands from one another. The command names in Nagios Core, such as `hostnames`, must be unique.
- It adds the `-p 8080` option to the command line call, specifying that when the call to `check_http` is made, the check will be made using the TCP port 8080, rather than the default value for TCP port 80.

The `check_http_altport` command can now be used as a check command in the same way a `check_http` command could be used. For example, a service definition that checks whether the `sparta.example.net` host is running an HTTP daemon on port 8080 might look something like this:

```
define service {
    use                generic-service
    host_name          sparta.example.net
    service_description HTTP_8080
    check_command       check_http_altport
}
```

## There's more...

This recipe's title implies that we should customize the existing commands by editing them in place and, indeed, this works fine if we really want to do things this way. Instead of copying the command definition, we could just add `-p 8080` or other customization to the command line and change the original command.

However, this is bad practice in most cases, mostly because it could break the existing monitoring and be potentially confusing to other administrators of the Nagios Core server. If we have a special case for monitoring (in this case, checking a nonstandard port for HTTP), then it's wise to create a whole new command based on the existing one with the customizations we need.

Particularly, if you share monitoring configuration duties with someone else on your team, changing the command could break the monitoring for anyone who had set up services using the `check_http` command before you changed it, meaning that their checks would all start failing because port 8080 would be checked instead.

There is no limit to the number of commands you can define, so you can be very liberal in defining as many alternative commands as you need. It's a good idea to give them instructive names that say something about what they do as well as to add explanatory comments to the configuration file. You can add a comment to the file by prefixing it with a `#` character:

```
#
# 'check_http_altport' command_definition. This is to keep track of the
# servers that have administrative panels running on an alternative port
# to confer special privileges to a separate instance of Apache HTTPD
# that we don't want to confer to the one for running public-facing
# websites.
#
define command {
    command_name    check_http_altport
    command_line    $USER1$/check_http -H $HOSTADDRESS$ -p 8080 $ARG1$
}
```

## See also

- The *Creating a new command* section in this chapter
- *Creating a new service*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*



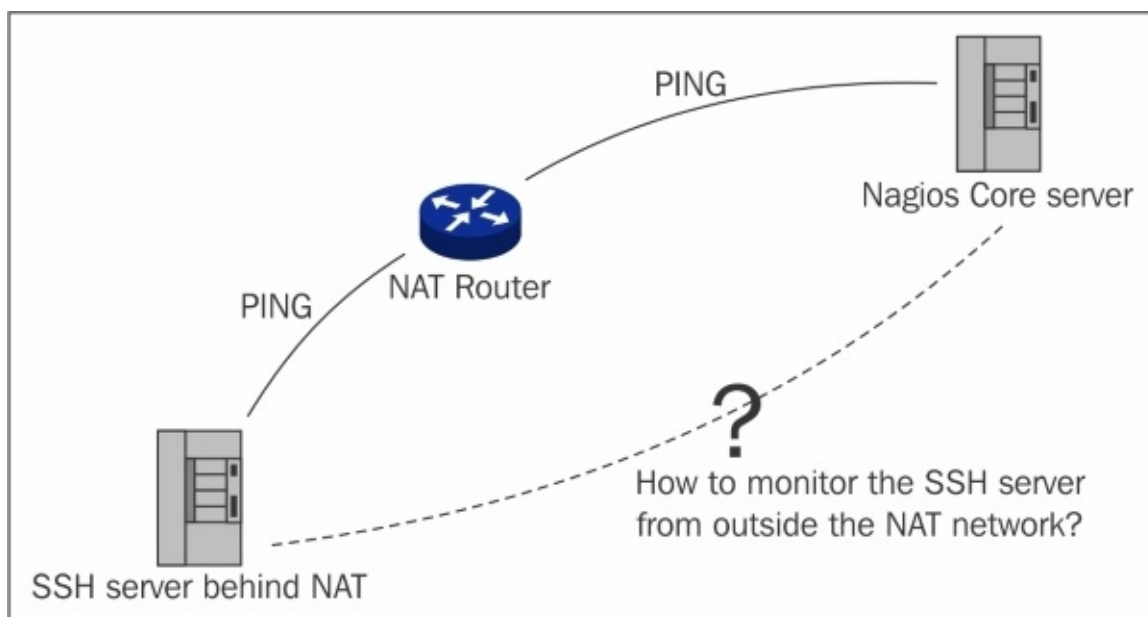
# Using an alternative check command for hosts

In this recipe, we'll learn how to deal with a slightly tricky case in network monitoring: that of monitoring a server that doesn't respond to PING but still provides some network service that requires checking.

It's good practice to allow PING where you can, as it's one of the stipulations in RFC 1122 and a very useful diagnostic tool not just for monitoring but for troubleshooting. However, sometimes, servers that are accessed only by a select few people might be configured not to respond to these messages, perhaps for reasons of secrecy. It's quite common for domestic routers to be configured this way.

Another very common reason for this problem, and the example we'll address here, is checking servers that are behind an IPv4 NAT firewall. It's not possible to address the host directly via an RFC1918 address such as 192.168.1.20 from the public Internet, and pinging the public interface of the router doesn't tell us whether the host for which it is translating addresses is actually working.

However, port 22 for SSH is forwarded from the outside to this server and it's this service that we need to check for availability. The structure of this part of the network, including the addressing problem, is in the following diagram:



We'll do this by checking whether the host is up through an SSH check, since we can't use PING on it from the outside like we normally would.

# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already. You should also be familiar with the relationship between services, commands, and plugins.



# How to do it...

We can specify an alternative check method for a host as follows:

1. Change to the directory containing the configuration file for the relevant host in Nagios Core. The default location for these files is `/usr/local/nagios/etc/objects`:
2. Find the file that contains the host definition for the host that won't respond to PING and edit it. In this example, our `crete.example.net` host is the one we want to edit:

```
# cd /usr/local/nagios/etc/objects
```

```
# vi crete.example.net.cfg
```

3. Change or define the `check_command` parameter of the host to the command that we want to use for the check instead of the usual `check-host-alive` or `check_ping` command. In this case, we want to use `check_ssh`. The resulting host definition might look something like this:

```
define host {  
    use                linux-server  
    host_name          crete.example.net  
    alias              crete  
    address            192.0.2.23  
    check_command      check_ssh  
}
```

Note that defining `check_command` still works even if we're using a host template such as `generic-host` or `linux-server`. It might also be good practice to check whether the host will actually respond to our check as we expect it to:

```
# sudo -s -u nagios  
$ /usr/local/nagios/libexec/check_ssh -H 192.0.2.23  
SSH OK - OpenSSH_6.7p1 Debian-5 (protocol 2.0) |  
time=0.012820s;;;0.000000;10.000000
```

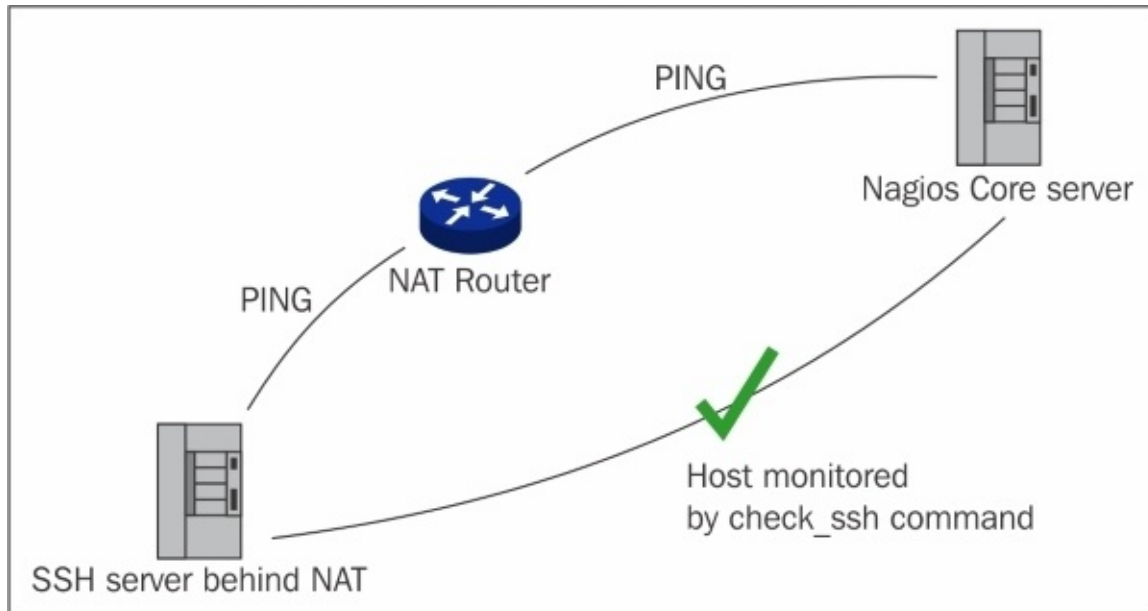
4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, the next scheduled host check for the `crete.example.net` server should show the host as UP because it was checked with the `check_ssh` command and not the usual `check-host-alive` command.

## How it works...

The configuration we added for the preceding `crete.example.net` host uses `check_ssh` to check whether the host is up, rather than using a check that uses PING. This is appropriate because the only public service accessible from `crete.example.net` is its SSH service. The following diagram demonstrates how the host is now being checked via SSH, despite not being accessible by PING:



The `check_ssh` command is normally used to check whether a service is available, rather than a host. However, Nagios Core allows us to use it as a host check command as well. Most service commands work this way; you could check a web server in the same way with `check_http`.

## There's more...

Note that for completeness' sake, it would also be appropriate to monitor the NAT router via PING or to perform an appropriate check on its public address. This way, if the host check for the SSH server fails, we can check whether the NAT router in front of it is still available, which assists in troubleshooting irrespective of whether the problem is with the server or with the NAT router. You can make this setup even more useful by making the NAT router a **parent** host for the SSH server behind it, which is explained in the *Creating a network host hierarchy* recipe in [Chapter 8](#), *Managing Network Layout*.

## See also

- *Monitoring SSH for any host, [Chapter 5](#), Monitoring Methods*
- *Checking an alternative SSH port, [Chapter 5](#), Monitoring Methods*
- *Monitoring local services on a remote machine with NRPE, [Chapter 6](#), Enabling Remote Execution*
- *Creating a network host hierarchy, [Chapter 8](#), Managing Network Layout*
- *Establishing a host dependency, [Chapter 8](#), Managing Network Layout*



# Writing a new plugin from scratch

Given the very useful standard plugins in the Nagios Plugins set and the large number of custom plugins available on Nagios Exchange, occasionally, as our monitoring setup grows more refined, we may well find that there is a service or property of a host that we would like to check. However, for this, there doesn't seem to be any suitable plugin available. Every network is different and, sometimes, the plugins that others have made by generously donating their time to the community don't quite cover all your bases. Generally, the more specific your monitoring requirements get, the less likely it is that there's a plugin available that does exactly what you need.

In this example, we'll deal with a very particular problem that we'll assume can't be dealt with effectively by any known Nagios Core plugins, and we'll write one ourselves using Perl. Here's the example problem.

Our Linux security team wants to be able to automatically check whether any of our servers are running kernels that have known exploits. However, they're not worried about every vulnerable kernel, only certain ones. They have provided us with the version numbers of three kernels, which have small vulnerabilities that they're not particularly worried about but that do need patching, and one they're extremely worried about.

Let's say, the minor vulnerabilities are in the kernels with version numbers 2.6.19, 2.6.24, and 3.0.1. The serious vulnerability is in the kernel with version number 2.6.39. Note that the version numbers in this case are arbitrary and don't necessarily reflect any real kernel vulnerabilities!

The team could log in to all the servers individually to check them, but the servers are of varying ages and access methods and are managed by different people. They would also have to manually check the kernel version more than once because it's possible that a naive administrator could upgrade the system to a kernel that's known to be vulnerable in an older release, and they also might want to add other vulnerable kernel numbers to be checked later on.

So, the team has asked us to solve the problem with Nagios Core monitoring and we've decided that the best way to do this is to write our own plugin, `check_vuln_kernel`, which checks the output of `uname(1)` for a kernel version string and then does the following:

- If the system is using one of the slightly vulnerable kernels, the plugin will return a `WARNING` state so that we can let the security team know that they should address it when they're next able to.
- If it's the highly vulnerable kernel version, it will return a `CRITICAL` state so that the security team knows that a patched kernel needs to be installed immediately.
- If `uname(1)` gives an error or output that we don't understand, it will return an `UNKNOWN` state, alerting the team to a bug in the plugin or possibly more serious problems with the server.
- Otherwise, it returns an `OK` state, confirming that the kernel is not known to be a vulnerable one.

- Finally, they want to be able to see at a glance in the Nagios Core monitoring what the kernel version is and whether it's vulnerable or not.

For the purposes of this example, we'll only monitor the Nagios Core server itself, but, via NRPE, we'd be able to install this plugin on the other servers that require this monitoring, where they'll work just as well. You should refer to the *Monitor local services on a remote machine with NRPE* recipe in [Chapter 6](#), *Enabling Remote Execution* to learn how to do this.

While this problem is very specific, we'll approach it in a very general way, which you'll be able to adapt to any solution, where it's required for a Nagios plugin to do the following tasks:

- Run a command and pull its output into a variable
- Check the output for the presence or absence of certain patterns
- Return an appropriate status based on those tests

All this means is that if you're able to do this, you'll be able to monitor anything from Nagios Core effectively!

# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already. You should also already be familiar with the relationship between services, commands, and plugins.

You should have Perl installed, at least version 5.10. This will include the required POSIX module. You should also have the Perl modules `Nagios::Plugin` (or `Monitoring::Plugin`) and `Readonly` installed. On Debian-like systems, you can install this using the following command:

```
# apt-get install libnagios-plugin-perl libreadonly-perl
```

On RPM-based systems, such as CentOS or Fedora Core, this should work:

```
# yum install perl-Nagios-Plugin perl-Readonly
```

This will be a rather long recipe that ties in a lot of Nagios Core concepts. For this, you should be familiar with all the following concepts:

- Defining new hosts and services and how they relate to one another
- Defining new commands and how they relate to the plugins they call
- Installing, testing, and using Nagios Core plugins

Some familiarity with Perl would also be helpful, but not required. We'll include comments to explain what each block of code is doing in the plugin.



# How to do it...

We can write, test, and implement our example plugin as follows:

1. Change to the directory containing the executable plugin programs for Nagios Core. The default location is `/usr/local/nagios/libexec`:

```
# cd /usr/local/nagios/libexec
```

2. Start editing a new file called `check_vuln_kernel`:

```
# vi check_vuln_kernel
```

3. Include the following code in it. Take note of the comments, which explain what each block of code is doing:

```
#!/usr/bin/env perl

# Use strict Perl style
use strict;
use warnings;
use utf8;

# Require at least Perl v5.10
use 5.010;

# Require a few modules, including Nagios::Plugin
use Nagios::Plugin;
use POSIX;
use Readonly;

# Declare some constants with patterns that match bad kernels
Readonly::Scalar my $CRITICAL_REGEX => qr/^2[.]6[.].39[^\d]/msx;
Readonly::Scalar my $WARNING_REGEX =>
    qr/^(?:2[.]6[.](?:19|24)|3[.]0[.].1)[^\d]/msx;

# Run POSIX::uname() to get the kernel version string
my @uname = uname();
my $version = $uname[2];

# Create a new Nagios::Plugin object
my $np = Nagios::Plugin->new();

# If we couldn't get the version, bail out with UNKNOWN
if ( !$version ) {
    $np->nagios_die('Could not read kernel version string');
}

# Exit with CRITICAL if the version string matches the critical pattern
if ( $version =~ $CRITICAL_REGEX ) {
    $np->nagios_exit( CRITICAL, $version );
}

# Exit with WARNING if the version string matches the warning pattern
if ( $version =~ $WARNING_REGEX ) {
    $np->nagios_exit( WARNING, $version );
}
```

```
}
```

```
# Exit with OK if neither of the patterns matched  
$np->nagios_exit( OK, $version );
```

4. Make the plugin that is owned by the nagios group and executable with chmod(1):

```
# chown root.nagios check_vuln_kernel  
# chmod 0770 check_vuln_kernel
```

5. Run the plugin directly to test it:

```
# sudo -s -u nagios  
$ ./check_vuln_kernel  
VULN_KERNEL OK: 3.16.0-4-amd64
```

We should now be able to use the plugin in a command and hence in a service check, just like any other command.

# How it works...

The code we added in the preceding new plugin file `check_vuln_kernel` can be understood in five general steps:

1. It runs Perl's POSIX `uname` implementation to get the version number of the kernel.
2. If that doesn't work, it exits with a `UNKNOWN` status.
3. If the version number matches anything in a pattern containing critical version numbers, it exits with a `CRITICAL` status.
4. If the version number matches anything in a pattern containing warning version numbers, it exits with a `WARNING` status.
5. Otherwise, it exits with an `OK` status.

It also prints the status as a string, along with the kernel version number, if it was able to retrieve one.

We might set up a command definition for this plugin as follows:

```
define command {  
    command_name    check_vuln_kernel  
    command_line    $USER1$/check_vuln_kernel  
}
```

In turn, we might set up a service definition for this command like this:

```
define service {  
    use                local-service  
    host_name          localhost  
    service_description VULN_KERNEL  
    check_command       check_vuln_kernel  
}
```

If the kernel was not vulnerable, the service's appearance in the web interface might be something like this:

|             |    |                     |             |     |                                 |
|-------------|----|---------------------|-------------|-----|---------------------------------|
| VULN_KERNEL | OK | 06-20-2015 18:44:10 | 0d 0h 0m 9s | 1/4 | VULN_KERNEL OK - 3.16.0-4-amd64 |
|-------------|----|---------------------|-------------|-----|---------------------------------|

However, if the monitoring server itself happened to be running a vulnerable kernel, it might look more like this (and send consequent notifications, if configured to do so):

|             |          |                     |              |     |                                       |
|-------------|----------|---------------------|--------------|-----|---------------------------------------|
| VULN_KERNEL | CRITICAL | 06-20-2015 18:48:06 | 0d 0h 0m 10s | 1/4 | VULN_KERNEL CRITICAL - 2.6.39-5-amd64 |
|-------------|----------|---------------------|--------------|-----|---------------------------------------|

## There's more...

This may be a simple plugin, but its structure can be generalized to all sorts of monitoring tasks. If we can figure out the correct logic to return the status we want in an appropriate programming language, then we can write a plugin to do basically anything.

A plugin like this could just as effectively be written in C for improved performance, but we'll assume, for simplicity's sake, that high performance for the plugin is not required and instead we can use a language that's better suited for quick ad hoc scripts like this one; in this case, Perl. The `utils.sh` file, also in `/usr/local/nagios/libexec`, allows us to write in the shell script if we'd prefer that.

If you prefer Python, the `nagiosplugin` library should meet your needs for both Python 2 and Python 3. Ruby users may like the `nagiosplugin` gem.

If you write a plugin that you think could be generally useful for the Nagios community at large, please consider putting it under a free software license and submitting it to Nagios Exchange so that others can benefit from your work. Community contribution and support is what has made Nagios Core such a great monitoring platform with such wide use.

Any plugin you publish in this way should conform to the Nagios Plugin Development Guidelines. At the time of writing this, these are available at <https://nagios-plugins.org/doc/guidelines.html>.

You may find older Nagios Core plugins written in Perl using the `utils.pm` file instead of `Nagios::Plugin` or `Monitoring::Plugin`. This will work fine, but `Nagios::Plugin` is recommended, as it includes more functionality out of the box and tends to be easier to use.

## See also

- *Monitoring local services on a remote machine with NRPE, [Chapter 6](#), Enabling Remote Execution*
- *The [Creating a new command](#) section in this chapter*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The [Customizing an existing command](#) section in this chapter*
- *The [Implementing threshold checks in a plugin](#) section in this chapter*
- *The [Using macros as environment variables in a plugin](#) section in this chapter*



# Implementing threshold checks in a plugin

You'll note that many of the plugins included in the Nagios Plugins set allow you to specify thresholds for different aspects of the tests that they perform, allowing custom configuration of which levels are ok, which need a warning, and which are critical. For example, the `check_ping` plugin requires us to specify thresholds with `-w` and `-c` options that define limits for round-trip-time and packet loss:

```
$ /usr/local/nagios/libexec/check_ping -H 192.0.2.21 -w 100,20% -c 200,40%
PING OK - Packet loss = 0%, RTA=0.20
ms|rta=0.200000ms;100.000000;100.000000;0.000000 pl=0%;10;20;0
```

In this case, the plugin's options are set only to raise a `WARNING` state if the round-trip-time for the check exceeds 100 milliseconds or if more than 20% of the packets are lost. It will raise a `CRITICAL` state if the check takes more than 200 milliseconds or 40% of the packets are lost.

When you're checking numeric values in a plugin, this is a useful way to allow the user to set their own thresholds for the check, rather than hardcoding them in the plugin itself. In this recipe, we'll adapt the plugin from the *Writing a new plugin from scratch* recipe to allow the user to set a threshold for kernel version numbers. We'll call this plugin `check_kernel_version`.

# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already, and you should already have successfully deployed the `check_vuln_kernel` plugin from the *Writing a new plugin from scratch* recipe in this chapter, including the installation of Perl and the two modules `Nagios::Plugin` (or `Monitoring::Plugin`) and `Readonly`.



# How to do it...

We can write, test, and implement our `check_kernel_version` plugin as follows:

1. Change to the directory containing the plugin binaries for Nagios Core. The default location is `/usr/local/nagios/libexec`:

```
# cd /usr/local/nagios/libexec
```

2. Start editing a new file called `check_kernel_version`:

```
# vi check_kernel_version
```

3. Include the following code in it. Take note of the comments, which explain what each block of code does:

```
#!/usr/bin/env perl
```

```
# Use strict Perl style
use strict;
use warnings;
use utf8;
```

```
# Require at least Perl v5.10
use 5.010;
```

```
# Require a few modules, including Nagios::Plugin
use Nagios::Plugin;
use POSIX;
use Readonly;
```

```
# Run POSIX::uname() to get the kernel version string
my @uname = uname();
my $version = $uname[2];
my ($version_major) = split m/[.]/msx, $version;
```

```
# Create a new Nagios::Plugin object
my $np = Nagios::Plugin->new( usage => 'Usage: %s -w THRESHOLD -c
THRESHOLD' );
```

```
# Add options allowing specifying warning and critical ranges
$np->add_arg(
    spec => 'warning|w=s',
    help => "-w, --warning=THRESHOLD\n"
        . '    Kernel version number threshold for returning warning',
    required => 1
);
$np->add_arg(
    spec => 'critical|c=s',
    help => "-c, --critical=THRESHOLD\n"
        . '    Kernel version number threshold for returning critical',
    required => 1
);
```

```
# Read options
$np->getopts();
```

```

# Compare the major version number to the thresholds
my $code = $np->check_threshold(
    check      => $version_major,
    warning    => $np->opts->warning,
    critical   => $np->opts->critical,
);

# Exit with the appropriate status
$np->nagios_exit( $code, $version );

# If we couldn't get the major version number, bail out with UNKNOWN
if ( !$version_major ) {
    $np->nagios_die('Could not read kernel version string');
}

```

4. Make the plugin that is owned by the nagios group and executable with `chmod(1)`:

```

# chown root.nagios check_kernel_version
# chmod 0770 check_kernel_version

```

5. Run the plugin directly to test it; your output may differ depending on your system's kernel version:

```

# sudo -s -u nagios
$ ./check_kernel_version -w 4: -c 3:
KERNEL_VERSION WARNING - 3.16.0-4-amd64

```

We should now be able to use the plugin in a command and hence in a service check, just like any other command.

# How it works...

The code in `check_kernel_version` differs from that of `check_vuln_kernel` in several ways:

- It tests only the major kernel version number, the very first part of the kernel version string.
- It allows us to specify the range of version numbers that should raise `WARNING` or `CRITICAL` states on the command line, rather than hardcoding them in the script. It does this using the `Nagios::Plugin` implementation of an option parser.
- It includes some basic output, showing usage and help information for the options. This is required by `Nagios::Plugin`.
- It uses the `check_threshold` method of the `Nagios::Plugin` package to check the version number against the thresholds for us.

We could have written our own code to compare the version numbers, but there's an advantage to using the `check_threshold` method; it uses the standard threshold format to specify the range for an alert level. In the recipe, we used these values:

- 4: For `-w`, this means that an alert is generated if the value being checked is less than 4.
- 3: For `-c`, this means an alert is generated if the value being checked is less than 3.

Because the major version number of the kernel in the recipe is 3, we got the `WARNING` output because it's less than 4, but not less than 3.

The threshold format syntax allows you to specify the ranges for alerts very carefully. There's a breakdown of the syntax available on the Nagios Plugins website, <https://nagios-plugins.org/doc/guidelines.html#THRESHOLDFORMAT>.

# There's more...

We might set up a command and corresponding service check for this new plugin as follows:

```
define command {
    command_name    check_kernel_version
    command_line    $USER1$/check_kernel_version -w $ARG1$ -c $ARG2$
}
define service {
    use                local-service
    host_name          localhost
    service_description    KERNEL_VERSION
    check_command       check_kernel_version!4:!3:
}
```

Note that we are able to define the WARNING and CRITICAL thresholds in the service definition as command arguments. This allows us to choose different thresholds for different services without editing the plugin's code.

Ideally, when writing plugins, we should include documentation and help output, particularly if we intend to distribute them to other users. The Nagios::Plugin module enforces some bare minimums, such as specifying the usage information output for the plugin if we declare any options, and requiring help information for each option.

If we run this plugin with a --help option, we get some useful output built by the Nagios::Plugin module's methods:

```
$ ./check_kernel_version --help
check_kernel_version
```

**This nagios plugin is free software, and comes with ABSOLUTELY NO WARRANTY. It may be used, redistributed and/or modified under the terms of the GNU General Public Licence (see <http://www.fsf.org/licensing/licenses/gpl.txt>).**

**Usage: check\_kernel\_version -w THRESHOLD -c THRESHOLD**

```
-?, --usage
    Print usage information
-h, --help
    Print detailed help screen
-V, --version
    Print version information
--extra-opts=[section][@file]
    Read options from an ini file. See http://nagiosplugins.org/extra-opts
    for usage and examples.
-w, --warning=THRESHOLD
    Kernel version number threshold for returning warning
-c, --critical=THRESHOLD
    Kernel version number threshold for returning critical
-t, --timeout=INTEGER
    Seconds before plugin times out (default: 15)
-v, --verbose
    Show details for command-line debugging (can repeat up to 3 times)
```

Note that one of these options is `--extra-opts`. The module implements this so we can put any options for the plugin call into a file if we wish. For example, we could put our `-w` and `-c` options into an INI file `check_kernel_version.ini`:

```
[check_kernel_version]
warning = 4:
critical = 3:
```

Then we could call it like this:

```
$ ./check_kernel_version --extra-opts=@check_kernel_version.ini
KERNEL_VERSION WARNING - 3.16.0-4-amd64
```

## See also

- The *Creating a new command* section in this chapter
- *Creating a new service*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- The *Writing a new plugin from scratch* section in this chapter
- The *Using macros as environment variables in a plugin* section in this chapter



# Using macros as environment variables in a plugin

The examples of commands we've seen so far have included all the information needed to execute the checks as command-line options, for example, specifying the host address for `check_ping` or the HTTP port for `check_http`. An alternative way to get the information required to execute checks is via **environment variables**. While this isn't enabled by default, it's a method of providing a large amount of context information to a plugin that may save some configuration effort. However, it's only suitable for specialized applications of Nagios Core and, because it's resource intensive, it may not scale very well, meaning it could be unsuitable for very busy Nagios Core servers.

In this example, we'll enable environment macros and then write a short wrapper script for `check_ping` that reads the address to PING from its executing environment, rather than from a command-line option.



# Getting ready

You should have a Nagios Core 4.0 or newer server running with a few hosts and services configured already, and you should have some understanding of how command objects define the usage of plugins and of the macros defined by Nagios Core.

This is a reasonably advanced recipe that assumes some basic familiarity with environment variables in Unix-like operating systems.

# How to do it...

We can enable environment macros and implement our command wrapper as follows:

1. Change to the directory containing the main configuration file for Nagios Core. The default location is `/usr/local/nagios/etc`:

```
# cd /usr/local/nagios/etc
```

2. Edit the `nagios.cfg` file:

```
# vi /usr/local/nagios/etc/nagios.cfg
```

3. Find the definition for the `enable_environment_macros` option. It defaults to `0`. Set it to `1` instead. Save the file and exit:

```
enable_environment_macros=1
```

4. Change to the directory containing the plugin binaries for Nagios Core. The default location is `/usr/local/nagios/libexec`:

```
# cd /usr/local/nagios/libexec
```

5. Edit a new file named `check_ping_hostaddress`:

```
# vi check_ping_hostaddress
```

6. Add the following code and then save the file and exit:

```
#!/bin/sh
exec /usr/local/nagios/libexec/check_ping -H "${NAGIOS_HOSTADDRESS:?}"
"$@"
```

7. Make the new file that is owned by the `nagios` group, and executable:

```
# chown root.nagios check_ping_hostaddress
# chmod 0770 check_ping_hostaddress
```

8. Test the plugin by running it without and with the required `NAGIOS_HOSTADDRESS` environment variable set:

```
# sudo -s -u nagios
$ ./check_ping_hostaddress -w 100,20% -c 200,40%
./check_ping_hostaddress: 2: ./check_ping_hostaddress:
NAGIOS_HOSTADDRESS: parameter not set or null
$ NAGIOS_HOSTADDRESS=127.0.0.1 ./check_ping_hostaddress -w 100,20% -c
200,40%
PING OK - Packet loss = 0%, RTA = 0.04
ms|rta=0.041000ms;100.000000;200.000000;0.000000 pl=0%;20;40;0
```

9. Go back to the root shell, validate the configuration, and reload Nagios Core:

```
$ exit
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

We should now have a `check_ping_hostaddress` plugin available to us, which will allow us to use it in a command without having to pass `$HOSTADDRESS$` in the `command_line` directive's value.

# How it works...

We enabled **macro environment variables** in the Nagios Core configuration. This makes the set of macros normally useable in the `command_line` directive, such as `$HOSTADDRESS$`, available to the script as environment variables, with their names prefixed with `NAGIOS_`.

We then wrote a **wrapper script** for the `check_ping` command that uses the `$NAGIOS_HOSTADDRESS$` environment variable to define the address it should check, meaning that we don't have to specify the target host on the command line. It also copies any other arguments given to it to the `check_ping` call with the `$@` expansion and throws an error if it doesn't have a value for `NAGIOS_HOSTADDRESS`.

We could add a command to use this plugin and a corresponding service check like this:

```
define command {
    command_name    check_ping_hostaddress
    command_line    $USER1$/check_ping_hostaddress -w 100,20% -c 200,40%
}
define service {
    use                local-service
    host_name          localhost
    service_description PING
    check_command       check_ping_hostaddress
}
```

We can get an idea of the set of variables passed to the plugin by having it dump its environment to a file somewhere within it to inspect its contents before the `exec` call:

```
export > /tmp/nagios-plugin-export
```

When the plugin is next executed by Nagios Core, we find a new file `/tmp/nagios-plugin-export` with contents that show the many macros now available to us as environment variables:

```
export LANG='en_US.UTF-8'
export LANGUAGE='en_US:en'
export NAGIOS_ADMINEMAIL='nagios@example.net '
export NAGIOS_ADMINPAGER='pagenagios@example.net '
export NAGIOS_ARG1=''
export NAGIOS_ARG10=''
export NAGIOS_ARG11=''
export NAGIOS_ARG12=''
...
```

Any of these variables can be used within the script. This includes the macros generated by any custom directives we might have declared for our host or service.

## There's more...

Not all the Nagios macros are exported this way to plugins by default, even with `enable_environment_macros` turned on. For a complete breakdown of the macros available to you on the command line and in the environment, we can consult the relevant sections of the Nagios Core documentation at the following links:

- <http://nagios.sourceforge.net/docs/nagioscore/4/en/macros.html>
- <http://nagios.sourceforge.net/docs/nagioscore/4/en/macrolist.html>

It's important to note that the documentation for `enable_environment_macros` cautions that making all the macros available as environment variables is reasonably resource intensive and therefore might be inappropriate for large setups. We should only enable this option after careful consideration of whether we need it. It may be preferable to pass the required values explicitly in the `command_line` directive, as the example configuration does by default.

## See also

- The *Creating a new command* section in this chapter
- *Creating a new service*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- The *Writing a new plugin from scratch* section in this chapter
- The *Implementing threshold checks in a plugin* section in this chapter
- *Using custom directives*, [Chapter 9](#), *Managing Configuration*



# Chapter 3. Working with Checks and States

In this chapter, we will cover the following recipes:

- Specifying how frequently to check a host or service
- Changing thresholds for PING RTT and packet loss
- Changing thresholds for disk usage
- Scheduling downtime for a host or service
- Managing brief outages with flapping
- Adjusting flapping percentage thresholds for a service



# Introduction

Once hosts and services are configured in Nagios Core, its behavior is primarily dictated by the checks it makes to ensure that hosts and services are operating as expected and, in turn, as a result of these checks, it concludes the state in which these hosts and services must be.

How often it's appropriate to check hosts and services and on what basis it's appropriate to flag a host or service as problematic depends very much on the nature of the service and the importance of it running all the time. If a host on the other side of the world is being checked with PING and, during busy periods, its round trip time is over 100 ms, this may not actually be a cause of concern at all and perhaps is not something to even flag a WARNING state over, let alone a CRITICAL one.

However, if the same host was on the local network where it would be appropriate to expect a round trip time of less than 10 ms, a round trip time of more than 100 ms could well be considered a grave cause for concern, perhaps signaling a packet storm or another problem with the local network. In such cases, we would want to notify the appropriate administrators immediately. Similarly, for services such as web servers, we may not be concerned by a response time of more than a second for a page on a busy, budget-shared webhost for customers. However, if the response time for a corporate website or a dedicated colocation customer was bad, it might be important to notify the web server administrator about it.

Not all hosts and services are, therefore, equal. Nagios Core provides several ways to define behaviors with more precision, which are as follows:

- How often a host or service should be checked with its appropriate `check_command`
- How bad a check's results have to be before a WARNING or CRITICAL problem is flagged, if at all
- Defining a **downtime** period for a host or service so that Nagios Core knows not to expect it to operate during a specified period of time, often for upgrades or other maintenance
- Whether to automatically tolerate flapping or hosts and services that seem to go up and down a lot

This chapter will use some common instances of problems with the preceding behaviors to give you examples that show how to configure them.



# Specifying how frequently to check a host or service

In this recipe, we'll adjust the definition of a very important host to ensure that it checks whether the host is up in every three minutes and, if it finds that the host is down as a result of the check failing, it will check again after a minute before it sends a notification about the state to its defined contact. We'll do this by customizing the definition for an existing host.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `sparta.example.net`, a host defined in its own file.

You should also understand the basics of commands and plugins, in particular the meaning of the `check_command` directive. These are covered in the recipes in [Chapter 2](#), *Working with Commands and Plugins*.

# How to do it...

We can customize the check frequency for a host as follows:

1. Change to the objects configuration directory for Nagios Core. The default location for the objects for the objects is `/usr/local/nagios/etc/objects`. If you've put the definition of your host in a different file, move it to its directory instead:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing your host definition and find the definition within the file:

```
# vi sparta.example.net.cfg
```

The host definition may look something like this:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
}
```

3. Add or edit the value of the `check_interval` directive to 3:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    check_interval    3
}
```

4. Add or edit the value of the `retry_interval` directive to 1:

```
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    check_interval     3
    retry_interval    1
}
```

5. Add or edit the value of `max_check_attempts` to 2:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    check_interval     3
    retry_interval     1
    max_check_attempts 2
}
```

6. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, Nagios Core will run the relevant `check_command` (probably something like `check-host-alive`) against this host every three minutes and, if it fails, will flag the host as down, check the same again after one minute, and only then send a notification to its defined contact if the second check fails as well.

## How it works...

The preceding configuration changed three properties of the host object type to effect the changes we needed:

- `check_interval`: This defines how long to wait between successive checks of the host under normal conditions. We set this to 3, or three minutes.
- `retry_interval`: This defines how long to wait between follow-up checks of the host after first finding problems with it. We set this to 1, or one minute.
- `max_check_attempts`: This defines how many total checks should we run before a notification is sent. We set this to 2 for two checks. This means that after the first failed check is run, Nagios Core will run another check a minute later and will only send a notification if this check fails as well. After two checks have been run and the host is still in a problem state, it will go from a `SOFT` state to a `HARD` state.

Note that setting these directives in a host that derives from a template, as is the case with our example, will override any of the same directives in the template.

## There's more...

It's important to note that we can also define the units used by the `check_interval` and `retry_interval` commands. They only use minutes by default, checking the `interval_length` setting that's normally defined in the root configuration file for Nagios Core, by default, `/usr/local/nagios/etc/nagios.cfg`:

```
interval_length=60
```

If we wanted to specify these periods in seconds instead, we could set this value to 1 instead of 60:

```
interval_length=1
```

This would allow us, for example, to set `check_interval` to 15, to check a host every 15 seconds. Note that if we have a lot of hosts with such a tight checking schedule, it might overburden the Nagios Core process, particularly if the checks take a long time to complete.

Don't forget that changing these properties for a large number of hosts can be tedious, so if it's necessary to set these directives to some common value for more than a few hosts, it may be appropriate to set the values in a **host template** and then have these hosts inherit from it. Refer to the *Using inheritance to simplify configuration* recipe in [Chapter 9, Managing Configuration](#), for more details. Note that the same three directives also work for service declarations and have the same meaning. We could define the same notification behavior for a service on `sparta.example.net` with a declaration like this:

```
define service {
    use                generic-service
    host_name          sparta.example.net
    service_description HTTP
    check_command       check_http
    address            192.0.2.21
    check_interval    3
    retry_interval    1
    max_check_attempts 2
}
```



## See also

- The *Scheduling downtime for a host* section in this chapter
- *Using inheritance to simplify a configuration*, [Chapter 9](#), *Managing Configuration*



# Changing thresholds for PING RTT and packet loss

In this recipe, we'll set up a service for a host that monitors PING and take a look at how to adjust the thresholds for the `WARNING` and `CRITICAL` states, which is done using **command arguments**. We'll accomplish this by setting up a service for an existing host that's already being checked with a `check_command`, such as `check-host-alive`. Our service will be used to monitor not whether the host is completely down, but whether it's responding to PING requests within a reasonable period of time.

This could be useful to notify and assist in diagnosing problems with the actual connectivity of a service or host.

This recipe will therefore serve as a good demonstration of the concepts of supplying arguments to a command and adjusting the `WARNING` and `CRITICAL` thresholds for a particular service.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already and have used a `check_command` of `check-host-alive`. We'll use the example of `sparta.example.net`, which is a host defined in its own file.

You should also understand the basics of how hosts and services fit together in a Nagios Core configuration and you should be familiar with the use of commands and plugins via the `check_command` directive.

# How to do it...

We can add our PING service to the existing host with custom round trip time and packet loss thresholds as follows:

1. Change to the objects configuration directory for Nagios Core. The default location is `/usr/local/nagios/etc/objects`. If you've put the definition of your host in a different file, move it to its directory instead:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file that contains your host definition:

```
# vi sparta.example.net.cfg
```

3. Add the following definition to the end of the file. The one value that interests us the most is the value of the `check_command` directive:

```
define service {
    use                generic-service
    host_name          sparta.example.net
    service_description PING
    check_command       check_ping!100,20%!200,40%
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, Nagios Core will not only run a host check of `check-host-alive` against your original host to ensure that it's up, but it will also run a more stringent check of the PING responses from the machine as a service to check whether it's adequately responsive:

- If the **round trip time (RTT)** of the PING response is greater than 100 ms (but less than 200 ms), Nagios Core will flag a **WARNING** state.
- If the RTT of the PING response is greater than 200 ms, Nagios Core will flag a **CRITICAL** state.
- If more than 20% (but less than 40%) of the PING requests receive no response, Nagios Core will flag a **WARNING** state.
- If more than 40% of the PING requests receive no response, Nagios Core will flag a **CRITICAL** state.

In both cases, a notification will be sent to the service's defined contacts if it is configured to do so. Otherwise, this service functions much the same as any other service and appears in the web interface as a service beneath the host.

## How it works...

The configuration we added to our existing host creates a new service with a `service_description` of `PING`. For `check_command`, we use the `check_ping` command, which uses the plugin of the same name. The interesting part here is what follows the `check_command` definition: the `!100, 20%!200, 40%` string.

In Nagios Core, the `!` character is used as a separator for arguments that should be passed to the command. In the case of `check_ping`, the first argument defines **thresholds** or conditions that, if met, should make Nagios Core flag a `WARNING` state for the service. Similarly, the second argument defines the thresholds for a `CRITICAL` state.

Each of the two arguments is comprised of two comma-separated terms: the first number is the threshold for the RTT of the `PING` request and its response that should trigger a state and the second number is the percentage of packet loss that should be tolerated before raising the same state.

This pattern of arguments is specific to `check_ping`; they would not work for other commands such as `check_http`.

# There's more...

If we want to look in a bit more detail at how these arguments are applied, we can inspect the command definition for `check_ping`. By default, this is in the `/usr/local/nagios/etc/objects/commands.cfg` file and looks as follows:

```
define command {
    command_name    check_ping
    command_line    $USER1$/check_ping -H $HOSTADDRESS$ -w $ARG1$ -c $ARG2$ -
p 5
}
```

In the value of `command_line`, the following four macros are used:

- `$USER1$`: This expands to `/usr/local/nagios/libexec` or the directory in which the Nagios Core plugins are normally kept, including `check_ping`.
- `$HOSTADDRESS$`: This expands to `host_name` for the host or service definition in which the command is used. In this case, it expands to `192.0.2.21`, the value of the address directive for the `sparta.example.net` host.
- `$ARG1$`: This expands to the value given for the first argument of the command; in our recipe's case, the `100,20%` string.
- `$ARG2$`: This expands to the value given for the second argument of the command; in our recipe's case, the `200,40%` string.

The complete command-line call for our specific check with all these substitutions made would therefore look like this:

```
/usr/local/nagios/libexec/check_ping -H 192.0.2.21 -w 100,20% -c 200,40% -p 5
```

This command line makes use of the following four parameters of the `check_ping` program:

- `-H`: This specifies the address of the host to be checked
- `-w`: This specifies the thresholds to raise a `WARNING` state
- `-c`: This specifies the thresholds to raise a `CRITICAL` state
- `-p`: This specifies the number of `PING` requests to be sent

We can run this directly from the command line on the Nagios Core server to see what the results of the check might be:

```
# /usr/local/nagios/libexec/check_ping -H 192.0.2.21 -w 100,20% -c 200,40% -p 5
```

This yields an output that includes the OK result of the check and also some performance data:

```
PING OK - Packet loss = 0%, RTA = 0.17
ms|rt=0.174000ms;100.000000;200.000000;0.000000 p1=0%;5;10;0
```

The arguments specified in the command are, therefore, used to customize the behavior of `check_command` for the particular host or service that is being edited.

The `check_ping` plugin isn't the only way you can monitor ICMP ECHO response times for a host; an alternative in the Nagios Plugins set is the `check_icmp` plugin. You may find this one more efficient to run on larger Nagios Core installations, as it requires `setuid` to run as root and thereby executes network code directly, rather than forking a `ping(8)` process and parsing its output. Its options and syntax are very similar to those of `check_ping`:

```
$ /usr/local/nagios/libexec/check_ping -H 192.0.2.21 -w 100,20% -c 200,40%
PING OK - Packet loss = 0%, RTA = 0.17
ms|rta=0.166000ms;100.000000;200.000000;0.000000 pl=0%;20;40;0
$ /usr/local/nagios/libexec/check_icmp -H 192.0.2.21 -w 100,20% -c 200,40%
OK - 192.0.2.21: rta 0.109ms, lost 0%|rta=0.109ms;100.000;200.000;0;
pl=0%;20;40;; rtmax=0.210ms;;; rtmin=0.082ms;;;;
```



## See also

- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The [Changing thresholds for disk usage](#) section in this chapter*
- *Customizing an existing command, [Chapter 2](#), Working with Commands and Plugins*
- *Creating a new command, [Chapter 2](#), Working with Commands and Plugins*
- *Monitoring PING for any host, [Chapter 5](#), Monitoring Methods*



# Changing thresholds for disk usage

In this recipe, we'll configure the Nagios Core server to check its own disk usage and to flag a `WARNING` or `CRITICAL` state depending on how little free space is left on the disk. We'll accomplish this by adding a new service to the already defined `localhost` host called `DISK`, which will run the `check_local_disk` command to examine the state of the mounted volumes on the server.

Because burgeoning disk usage can creep up on any system administrator and because of the dire effect it can have when a disk suddenly gets filled completely without any warning, this is among the more important things to monitor in any given network.

For simplicity, we'll demonstrate this only for the monitoring server itself, as a host called `localhost` on `127.0.0.1`. This is because the `check_disk` plugin can't directly check the disk usage of a remote server. However, the principles discussed here could be adapted to run the check on a remote server using `check_nrpe`. The use of NRPE and alternative methods to run remote checks are discussed in all the recipes in [Chapter 6](#), *Enabling Remote Execution*.

# Getting ready

You should have a Nagios Core 4.0 or newer server with a definition of `localhost` so that the monitoring host is able to check itself. A host definition for `localhost` is included in the sample configuration located at `/usr/local/nagios/etc/objects/localhost.cfg`. You should also understand the basics of how hosts and services fit together in a Nagios Core configuration and be familiar with the use of commands and plugins via the `check_command` directive.

We'll use the example of `olympus.example.net` as our Nagios Core server. We'll have it check its own disk, with its mount point at `/`.

# How to do it...

We can add our DISK service to the existing host with custom usage thresholds as follows:

1. Change to the object's configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move it to its directory instead:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file that contains your host definition:

```
# vi localhost.cfg
```

3. Add the following definition to the end of the file. The one value that interests us the most here is the value of the `check_command` directive:

```
define service {  
    use                local-service  
    host_name          localhost  
    service_description DISK  
    check_command       check_local_disk!10%!5%!/  
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, a new service is created for the `localhost` host that checks the disk usage on `/` and flags a `WARNING` state for the service if the free space is below 10% and a `CRITICAL` state if the free space is below 5%.

In both the cases, a notification will be sent to the service's defined contacts, if configured to do so.

## How it works...

The configuration we added for our existing host creates a new service with a `service_description` of `DISK`. For the `check_command`, we use the `check_local_disk` command, which in turn uses the `check_disk` plugin to check the local machine's disks. The interesting part here is what follows the `check_local_disk` definition: the `!10%!5%!/` string.

In Nagios Core, the character `!` is used as a separator for arguments that should be passed to the command. In the case of `check_local_disk`, the first two arguments define thresholds or conditions that, if met, should make Nagios Core flag a `WARNING` state (the first argument, 10%) or a `CRITICAL` state (the second argument, 5%) for the service. The third argument defines the mount point of the disk to check, `/`. If you prefer, you can instead use its device name, for example `/dev/sda1`.

# There's more...

If we want to look in a bit more detail at how these arguments are applied, we can inspect the command definition of `check_local_disk`. By default, this is in the `/usr/local/nagios/etc/objects/commands.cfg` file and looks as follows:

```
define command {  
    command_name    check_local_disk  
    command_line    $USER1$/check_disk -w $ARG1$ -c $ARG2$ -p $ARG3$  
}
```

Note that in this case, the `command_name` value and the name of the plugin used in the `command_line` are not the same.

In the value for `command_line`, the following four macros are used:

- `$USER1$`: This expands to `/usr/local/nagios/libexec` or the directory in which the Nagios Core plugins are normally kept, including `check_disk`.
- `$ARG1$`: This expands to the value given for the first argument of the command; in this case, the `10%` string.
- `$ARG2$`: This expands to the value given for the second argument of the command; in this case, the `5%` string.
- `$ARG3$`: This expands to the value given for the third argument of the command; in this case, the `/` string.

The complete command-line call for our specific check with all these substitutions made would therefore look like this:

```
/usr/local/nagios/libexec/check_disk -w 10% -c 5% -p /
```

This command line makes use of the following three parameters of the `check_disk` program:

- `-w`: This specifies the thresholds to raise a `WARNING` state
- `-c`: This specifies the thresholds to raise a `CRITICAL` state
- `-p`: This specifies the mount point or device file of the disk to be checked

We can run this directly from the command line on the Nagios Core server to see what the results of the check might be:

```
# /usr/local/nagios/libexec/check_disk -w 10% -c 5% -p /
```

The output includes both the OK result of the check and also some performance data:

```
DISK OK - free space: / 2575 MB (71% inode=78%);| /=1044MB;3432;3051;0;3814
```

## See also

- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The *Changing thresholds for PING RTT and packet loss* section in this chapter*
- *Customizing an existing command, [Chapter 2](#), Working with Commands and Plugins*
- *Creating a new command, [Chapter 2](#), Working with Commands and Plugins*
- *Monitoring PING for any host, [Chapter 5](#), Monitoring Methods*





# Scheduling downtime for a host or service

In this recipe, we'll learn how to schedule downtime for a host or service in Nagios Core. This is useful to elegantly suppress notifications for some predictable period of time; a very good example of this is when servers require downtime to be upgraded or to have their hardware checked.

In this example, we'll demonstrate how to schedule downtime for a host named `sparta.example.net`, and we'll examine the changes it makes in the web interface.

# Getting ready

You should have a Nagios Core 4.0 or newer server with a definition for at least one host, at least one service, and some idea of when you would like your downtime to be scheduled. You should also have a working web interface, as per the standard installation of Nagios Core 4.0.

You should also have Nagios Core configured to process external commands and have given your web interface user the permissions to apply them. If you are logging in as the nagiosadmin user as per the recommended quick start guide, you can check this is working by ensuring the following directive in `/usr/local/nagios/etc/nagios.cfg` is set to 1:

```
check_external_commands=1
```

Permissions to submit external commands from the web interface are defined in `/usr/local/nagios/etc/cgi.cfg`; verify that your username is included in these directives:

```
authorized_for_all_service_commands=nagiosadmin  
authorized_for_all_host_commands=nagiosadmin
```

If you have followed the Nagios Core quick start guide, you will probably find that external commands are already being accepted and working (<http://nagios.sourceforge.net/docs/nagioscore/4/en/quickstart.html>).

# How to do it...

We can set up a fixed period of scheduled downtime for our host and service as follows:

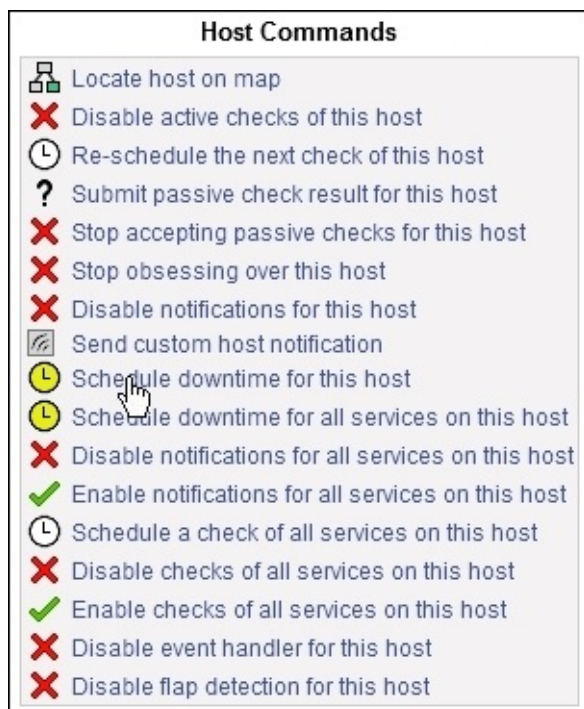
1. Log in to the web interface for Nagios Core.
2. Click on **Hosts**, which is on the left-hand side menu.



3. Click on the host's name in the table that comes up to view the details for that host.

|                    |      |    |                     |
|--------------------|------|----|---------------------|
| pop3.example.net   | POP3 | OK | 06-28-2015 16:00:05 |
| smtp.example.net   | SMTP | OK | 06-28-2015 16:02:35 |
| sparta.example.net | HTTP | OK | 06-28-2015 15:58:05 |

4. Click on **Schedule downtime for this host** on the **Host Commands** menu.



5. Fill out the fields in the resulting form. Include the following details:

- **Host Name:** This is the name of the host for which you're scheduling downtime. This should will be filled out for you.
- **Author:** This refers to your name, for records regarding who scheduled the downtime. This may be grayed out and may state Nagios Admin; that's fine.
- **Comment:** This refers to a comment that explains the reason for the downtime.
- **Start Time:** This is the time at which the scheduled downtime should begin and state notifications end.
- **End Time:** This refers to the time at which the scheduled downtime should end and state notifications resume.

**Command Options**

Host Name:

Author (Your Name):

Comment:

Triggered By:

Start Time:

End Time:

Type:

If Flexible, Duration:  Hours  Minutes

Child Hosts:

In this case, our downtime will be from 4.00 pm to 6.00 pm on June 28, 2015. Click on **Commit** to submit the downtime definition and then on **Done** in the screen that follows.

With this done, we can safely bring the `sparta.example.net` host down between the nominated times and any notifications for the host and any of its services will be suppressed until the downtime is over.

Note that restarting Nagios Core is not required for this step, as it would be for changes made to Nagios Core's configuration files. The change is done on the fly.

Note also that comments now appear in the detailed information for both the host and service, defining the downtime and including the reason specified for it.

## How it works...

The preceding steps nominate a period of downtime for both the `sparta.example.net` server and all of its services. This accomplishes two things:

- It suppresses all notification e-mails for the host or service for the appropriate time period, including `RECOVERY` notifications. The only exceptions are `DOWNTIMESTART` and `DOWNTIMEEND` notifications.
- It adds a comment to the host or service that shows the scheduled downtime for the benefit of anyone else who might be using the web interface.

Nagios Core keeps track of any downtime defined for all the hosts and services and prevents the notifications it would normally send out during that time. Note that it will still run its checks and record the state of both the hosts and services even during downtime. All that is suppressed are the notifications, not the actual checks.

## There's more...

Note that downtime for individual services can be applied in much the same way, by clicking on **Schedule downtime for this service** in the web interface, which is under **Service Commands**.

What was defined in this recipe was a method to define **fixed downtime**, where we know ahead of time when the host or its services are likely to be unavailable. If we don't actually know what time the unavailability will start, but we do know how long it's likely to last, we can define flexible downtime. This means that the downtime can start any time within the nominated period and it will last for the length of time we specify from that point.

A notification event is also fired when the host or service begins downtime, called DOWNTIMESTART, and another is fired when the downtime ends, called DOWNTIMEEND. This may be a useful notification to send to the relevant contact or contact group if they'd like to be notified when this happens. This can be arranged by ensuring that the host or service is configured to send these messages, by including the `s` flag in the `notification_options` directive for both hosts and services and correspondingly in the contact definition:

```
notification_options    d,u,r,f,s
```

## See also

- The *Managing brief outages with flapping* section in this chapter
- The *Adjusting flapping percentage thresholds for a service* section in this chapter
- *Choosing states for notification*, [Chapter 4](#), *Configuring Notifications*
- *Tolerating a certain number of failed checks*, [Chapter 4](#), *Configuring Notifications*
- *Adding comments on hosts or services in a web interface*, [Chapter 7](#), *Using the Web Interface*





# Managing brief outages with flapping

In this recipe, we'll learn how to use Nagios Core's state flapping detection and handling techniques to avoid sending excessive notifications when a host or service changes its state too frequently. This is useful in circumstances where a host or service is changing between OK to WARNING to CRITICAL states too frequently within the last 21 checks; if the percentage of state changes is too high, then Nagios Core will suppress further notifications and add a comment to the host or service, showing that it is flapping.

Flap detection is normally enabled by default in Nagios Core and it is part of the recommended `generic-host` host template and the `generic-service` service template. It's therefore likely that it's already enabled on most servers and we only need to check that it's still working.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host and one service configured already. You should also have access to a working web interface for the Nagios Core server. It would be helpful if you are monitoring a test service that you can bring up and down to trigger the flap detection.

You should be familiar with the way that hosts and services change states, as a result of their checks, and the different states corresponding to hosts and services, to understand the basics of how flap detection works.

# How to do it...

We can check whether flap detection is enabled for our Nagios Core server, our hosts, and our services as follows:

1. Change to the configuration directory for Nagios Core. The default location is `/usr/local/nagios/etc`:

```
# cd /usr/local/nagios/etc
```

2. Edit the `nagios.cfg` file.

```
# vi nagios.cfg
```

3. Look for an existing definition for the `enable_flap_detection` directive and verify that it is set to 1:

```
enable_flap_detection=1
```

4. If this was not set to 1, after we've changed it, we will probably also need to at least temporarily disable the `use_retained_program_state` directive in the same file:

```
use_retained_program_state=0
```

5. Edit the file for our particular hosts/services. We should verify that at least one of the following is the case:

The host or service inherits from a template that has the `enable_flap_detection` directive set to 1. For example, both the `generic-host` and `generic-service` templates defined by default in `/usr/local/nagios/objects/templates.cfg` do this.

The host or service itself has the `enable_flap_detection` directive set to 1 in its own definition.

In the latter case, the configuration for the host or service might look as follows:

```
define host {
    ...
    flap_detection_enabled 1
}
define service {
    ...
    flap_detection_enabled 1
}
```

6. If any of the preceding configuration was changed, validate the new configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

7. Verify that, for the hosts or services for which flap detection is wanted, the word **ENABLED** appears in the details for that host or service:

|                 |         |
|-----------------|---------|
| Active Checks:  | ENABLED |
| Passive Checks: | ENABLED |
| Obsessing:      | ENABLED |
| Notifications:  | ENABLED |
| Event Handler:  | ENABLED |
| Flap Detection: | ENABLED |

8. When you are done with the preceding steps, if a host or service changes its state too frequently within 21 checks, it will be flagged as flapping and will appear with a custom icon in the views of that host or service:

|                                         |      |                                                                                   |    |                     |
|-----------------------------------------|------|-----------------------------------------------------------------------------------|----|---------------------|
| sparta.example.net                      | HTTP |  | OK | 06-28-2015 16:30:07 |
| This service is flapping between states |      |                                                                                   |    |                     |

9. A comment is also placed in the host or service, explaining what happened for the benefit of anyone who views the host or service in the web interface, who might, perhaps, be wondering why the notifications have stopped:

| Service Comments                                                                                                                                                                                              |                  |                                                                                                                                                                                                                                                             |            |            |                |         |                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|------------|----------------|---------|---------------------------------------------------------------------------------------|
|  Add a new comment  Delete all comments |                  |                                                                                                                                                                                                                                                             |            |            |                |         |                                                                                       |
| Entry Time                                                                                                                                                                                                    | Author           | Comment                                                                                                                                                                                                                                                     | Comment ID | Persistent | Type           | Expires | Actions                                                                               |
| 06-28-2015 16:30:07                                                                                                                                                                                           | (Nagios Process) | Notifications for this service are being suppressed because it was detected as having been flapping between different states (24.2% change >= 20.0% threshold). When the service state stabilizes and the flapping stops, notifications will be re-enabled. | 1          | No         | Flap Detection | N/A     |  |

10. There will also be an indicator on the details for the host or service defining whether the host or service is flapping:

|                           |                           |
|---------------------------|---------------------------|
| Is This Service Flapping? | YES (23.16% state change) |
|---------------------------|---------------------------|

11. For hosts or services that don't have flap detection enabled, this particular field simply reads **N/A**, and the **Flap Detection** field below it will display **DISABLED**.

## How it works...

The logic behind determining flap detection is actually quite complex. For our purposes, it suffices to explain flap detection as being based on whether a host or service has changed its state within its last 21 checks too often—with the thresholds usually expressed as a percentage.

This is discussed in great detail, including the formulas that are used to determine the flapping state in the Nagios Core 4.0 documentation, which is available online at <http://nagios.sourceforge.net/docs/nagioscore/4/en/flapping.html>.

## There's more...

A common cause of flapping is that checks are too stringent. As an example, if you are checking a busy shared web server's response time is less than 50 ms, while the server is busy checks might pass and fail without actually giving an accurate reflection of whether the service is doing its job. In this case, it would be appropriate to loosen the thresholds of the service by increasing its percentage thresholds so that it isn't quite so ready to flag a WARNING or CRITICAL state over things that aren't actually very worrisome. Flap detection can help diagnose these sorts of cases.

We can also enable or disable flap detection for a host via the web interface; in the details screen for both hosts and services, a menu item is available under **Host Commands**, which is labeled **Enable/Disable flap detection for this host**, and under **Service Commands** there's another item labeled **Enable/Disable flap detection for this service**.

These may be useful when we want to turn flap detection on or off for a particular host or service temporarily, perhaps because under certain circumstances it is or is not appropriate to use the feature. For a permanent setup and for clarity, it would be best to include it explicitly in the configuration as shown in this recipe.

## See also

- The *Adjusting flapping percentage thresholds for a service* section in this chapter
- *Tolerating a certain number of failed checks*, [Chapter 4](#), *Configuring Notifications*
- *Adding comments on hosts or services in a web interface*, [Chapter 7](#), *Using the Web Interface*





# Adjusting flapping percentage thresholds for a service

In this recipe, we'll learn how to adjust the percentage thresholds for a host's or service's flap detection. This means that we can adjust how frequently a host or service has to change its state within its last 21 checks before Nagios Core concludes it's flapping and suppresses notifications until its state becomes stable again.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host and one service configured already. You should also have access to a working web interface for the Nagios Core server.

You should be familiar with the way hosts and services change their state as a result of their checks and the different states corresponding to hosts and services, to understand the basics of how flap detection works. Flap detection should also already be enabled and working for the appropriate hosts and services.

# How to do it...

We can adjust the thresholds for flap detection for a specific host or service as follows:

1. Change to the objects configuration directory for Nagios Core. The default location is `/usr/local/nagios/etc/objects`:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the host or service for which we want to set the thresholds:

```
# vi sparta.example.net.cfg
```

3. Within the host or service definition, set the `low_flap_threshold` and/or `high_flap_threshold` values to appropriate percentages:

```
define host {  
    ...  
    high_flap_threshold 50.0  
    low_flap_threshold 25.0  
}  
define service {  
    ...  
    high_flap_threshold 45.0  
    low_flap_threshold 20.0  
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, the flapping thresholds for the host or service should be changed appropriately for future checks.

# How it works...

The preceding configuration changes include the following directives:

- `high_flap_threshold`: This refers to a host or service that changes its state a certain percentage of the time must exceed this percentage threshold before it is determined to be flapping
- `low_flap_threshold`: If a host or service is already in the flapping state, its state change percentage must fall to this amount or fall below this percentage threshold before the flapping state will end

## Note

For a detailed breakdown of how the state change percentage is calculated, refer to the Nagios Core 4.0 documentation online at

<http://nagios.sourceforge.net/docs/nagioscore/4/en/flapping.html>.

## There's more...

If appropriate, we can also set a global default for hosts and services' flap thresholds with the following directives in `/usr/local/nagios/etc/nagios.cfg`. A set of possible values for these attributes might be:

```
low_service_flap_threshold=5.0  
high_service_flap_threshold=20.0  
low_host_flap_threshold=5.0  
high_host_flap_threshold=20.0
```

These values correspond to percentages of state changes in the same way that the per-host and per-service configurations do.

Note that there are separate directives for hosts and services in this case. These values are also overridden if you specify thresholds for a particular service or host, as we did in this recipe.

## See also

- The *Managing brief outages with flapping* section in this chapter





# Chapter 4. Configuring Notifications

In this chapter, we will cover the following topics:

- Configuring notification periods
- Configuring notifications for groups
- Choosing states for notification
- Specifying the number of failed checks before notification
- Automating contact rotation
- Defining an escalation for repeated notifications
- Defining a custom notification method
- Filtering notifications based on a host or service value

# Introduction

Notifications in Nagios Core refer to the events fired and the messages generated when hosts or services change state, for the purpose of informing appropriate people or systems.

For example, when connectivity to a host is lost, it leaves the UP state and goes to DOWN state on the next check. This eventually generates a notification event and, provided the appropriate flags are set and a contact is available, a message about the state change is generated and dealt with in some way defined in the configuration.

The default text for such notifications goes into some detail about the problem, including the output from the appropriate plugin command:

```
***** Nagios *****  
Notification Type: PROBLEM  
Host: sparta.example.net  
State: DOWN  
Address: 192.0.2.21  
Info: CRITICAL - Host Unreachable (192.0.2.21)  
Date/Time: Sat Jul 13 21:45:09 NZST 2015
```

So with this kind of text generated as a result of the event, the next question is what Nagios Core should do with it. A common method of notification that will be familiar to most Nagios Core administrators is sending this text as an e-mail message using the system mailer, but the same generated message can be used in a wide variety of ways. Just as commands can be flexibly defined in terms of the command lines they run, the action appropriate for a particular contact can be defined very flexibly to use the text of any notifications that the contact is configured to receive.

In this chapter, we'll learn how to refine notifications management in Nagios Core, to make sure that the appropriate people or systems are notified about appropriate network events and not nagged about others that may not matter. We'll also learn how to set up notification methods beyond the simple e-mail message and how to escalate notifications when problems remain unfixed after a certain period of time.



# Configuring notification periods

In this recipe, we'll adjust the configuration for a service that has been bugging us with notifications late at night. We'll arrange to keep checking this host, `sparta.example.net`, on a 24x7 basis, but we'll prevent it from sending notifications outside of work hours using two of the predefined time periods in the default Nagios Core configuration.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `sparta.example.net`, a host defined in its own file.

# How to do it...

We can define the `check_period` and `notification_period` for our host as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to its directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing your host definition and find the definition within the file:

```
# vi sparta.example.net.cfg
```

The host definition may look something like this:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address             192.0.2.21
}
```

3. Add or edit the value for the `check_period` directive to `24x7`:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address             192.0.2.21
    check_period       24x7
}
```

4. Add or edit the value for the `notification_period` directive to `workhours`:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address             192.0.2.21
    check_period       24x7
    notification_period workhours
}
```

5. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, Nagios Core will run its check of the host all the time (24 hours a day, 7 days a week), including logging UP and DOWN states, showing them in the web interface, and reporting, but will only send notifications to the nominated contact for the host during work hours, that is from 9 a.m. to 5 p.m., Monday to Friday, by default.

# How it works...

The preceding configuration changed two properties of the host object type to effect the changes we needed:

- `check_period` : This property defines the time periods during which the host should be checked; we chose the predefined period `24x7`, meaning the checks against the host are always to be run, regardless of the time of day or night.
- `notification_period` : This property defines the time periods during which the host should send notifications to the appropriate contact; we chose the predefined period `workhours`, meaning that notifications will only be sent from 9 a.m. to 5 p.m., Monday to Friday.

Exactly the same configuration could be applied to a service, rather than a host:

```
define service {
    ...
    check_period      24x7
    notification_period workhours
}
```

The two time periods to which this new configuration refers are themselves defined in the Nagios Core configuration. The ones we've used, `24x7` and `workhours`, are standard time periods included in the defaults, kept in the file

`/usr/local/nagios/etc/objects/timeperiods.cfg`. The definition for `workhours`, for example, looks like this:

```
define timeperiod {
    timeperiod_name workhours
    alias           Normal Work Hours
    monday          09:00-17:00
    tuesday          09:00-17:00
    wednesday        09:00-17:00
    thursday         09:00-17:00
    friday           09:00-17:00
}
```

The ones we've used here are just examples of common time periods. We can define time periods ourselves with great precision; see the recipe *Creating a new time period* in [Chapter 1, Understanding Hosts, Services, and Contacts](#) for details on how to do this.

## There's more...

The distinction between the two directives is very important. Under most circumstances, we will want a `check_period` that corresponds to all the times for services that are up day and night, because it means we can retain information about the long-term uptime for the service. Keeping a distinct `notification_period` simply allows us to control when notifications should be sent, perhaps because they would otherwise go to a pager and wake up a grumpy systems administrator for a relatively unimportant system!

It's valid Nagios Core configuration to set a `check_period` for a reduced time period, for example, `workhours`. However, you shouldn't need to do this unless your host is actually going to be down outside these hours. In most cases, it's appropriate to use the 24x7 time period as a value for the `check_period`.



## See also

- *Creating a new time period*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Specifying how frequently to check a host or service*, [Chapter 3](#), *Working with Checks and States*
- The *Configuring notification for groups* section in this chapter



# Configuring notifications for groups

In this recipe, we'll learn how to define a contact group for a host. We'll demonstrate this as part of the general best practice of sending notifications to contact groups, rather than to individual contacts. This allows for more flexibility in assigning individual contacts to the appropriate groups for receiving appropriate notifications.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `sparta.example.net`, a host defined in its own file, which we'll configure to send its notifications to an existing contact group called `noc`.

# How to do it...

We can configure notifications to be sent to our contact group as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. The notifications will need to be directed to a contact group object. We will need to ensure that an appropriate `contact_group` definition exists for this. A valid definition might look something like the following, perhaps kept in `contacts.cfg`:

```
define contactgroup {
    contactgroup_name    noc
    alias                Network Operations
    members              john,jane
}
```

3. If any contacts are referenced for the group, you should ensure those are also defined. They might look something like this:

```
define contact {
    use                generic-contact
    contact_name       john
    alias              John Smith
    email              john@example.net
}
define contact {
    use                generic-contact
    contact_name       jane
    alias              Jane Doe
    email              jane@example.net
}
```

4. Edit the file containing your host definition and find the definition within the file:

```
# vi sparta.example.net.cfg
```

The host definition may look something like this:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    notification_period 24x7
}
```

5. Add or edit the value for the `contact_groups` directive to `noc`:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    contact_groups     noc
}
```

```
    address            192.0.2.21
    notification_period 24x7
    contact_groups     noc
}
```

6. If the contacts directive for the host is already defined, you may wish to remove it, though you don't have to.
7. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, all notifications for this host should be sent to each member of the noc group, in our example, the contacts jane and john.

## How it works...

Defining the contacts that should receive notifications for particular hosts can be done by referring to them individually, as groups, or both. We could have managed the same result as the preceding one by defining the contacts directive for the host as jane, john, referring to the individual contacts directly rather than by their group name, with the two names separated by a comma.

Much the same configuration process applies for setting the contact group for services, rather than hosts; we add a value for the contact\_groups directive to the service definition:

```
define service {  
    ...  
    contact_groups    noc  
}
```

When an event takes place for the host or service that prompts a notification and the valid notification\_option flags are set for that event, Nagios Core will check the values of both the contacts and the contact\_groups directives for the host to determine whom to send the notification to. In our case, the server finds the value of contact\_groups to be noc and so refers to that contact group to identify the individual contacts within it; it then sends the notification to each of them.

## There's more...

Which of the two methods to use for defining contacts for hosts or services, whether by individual contact references or groups, is up to you. For large numbers of hosts and services, it tends to be easier to direct notifications to the appropriate groups rather than individuals, as it affords us more flexibility in choosing who the notifications should be sent to by simply changing the individuals in the group.

Note that we can use both methods if necessary; if for a particular host or service it would be appropriate to also send a notification to an individual contact named `james`, we could define the host as follows:

```
define host {  
    ...  
    contact_groups    noc  
    contacts         james  
}
```

You need to define at least one contact or contact group for a host or service, or have it inherit one through a template, for it to be a valid host configuration for Nagios Core.



## See also

- The *Configuring notification periods* section in this chapter
- *Creating a new contact*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new contact group*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- The *Choosing states for notification* section in this chapter
- The *Automating contact rotation* section in this chapter



# Choosing states for notification

In this recipe, we'll learn how to refine which notifications are sent by a host or service and which notifications should be received by a particular contact. We'll do this by changing both the notification types that hosts and services should generate and, to complement that, the notification types that a contact should be configured to receive.

# Getting ready

You should have a Nagios Core 4.0 or newer server, with at least one host configured already. We'll use the example of `sparta.example.net`, a host defined in its own file, which we'll configure to send only the `DOWN` and `RECOVERY` notifications, ignoring other notifications such as `WARNING` and `UNKNOWN`. It will send the notifications to an existing contact called `john`; we'll ensure that this contact is configured to receive those notifications.

# How to do it...

We can configure the notification types for our host as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for your host. It might look something like this:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    contacts           john
    notification_period 24x7
}
```

3. Add a value for the `notification_options` directive. In our case, we'll use `d,r`, to correspond to the notifications for the DOWN and RECOVERY states.

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    contacts           john
    notification_period 24x7
    notification_options d,r
}
```

4. We should also ensure that the notifications for this host are enabled; specifically, that `notifications_enabled` is set to 1. An easy way to set this and other relevant directives is to have the host inherit from a template like `linux-server`, as done previously, but we can set it explicitly if we prefer:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    contacts           john
    notification_period 24x7
    notification_options d,r
    notifications_enabled 1
}
```

5. Edit the file containing the definition for the host's nominated contacts, or the contacts within its nominated contact group. In this case, we're editing the host's nominated contact, `john`. The definition might look something like the following:

```
define contact {
```

```

        use                generic-contact
        contact_name      john
        alias              John Smith
        email              john@example.net
    }

```

6. Edit or set the directive for `host_notification_options` for each contact to include the same values that you set for the host. In this case, we set it to `d,u,r`, which means that John Smith can receive the `DOWN`, `UNREACHABLE`, and `RECOVERY` notifications about the hosts.

```

define contact {
    use                generic-contact
    contact_name      john
    alias              John Smith
    email              john@example.net
    host_notification_options  d,u,r
}

```

7. Again, you should ensure that the contacts are configured to receive host notifications, with the `host_notifications_enabled` directive set to `1`. We can do this via inheritance from a template like `generic-contact` as done in the preceding case, which tends to be easier, or we can set it explicitly:

```

define contact {
    use                generic-contact
    contact_name      john
    alias              John Smith
    email              john@example.net
    host_notification_options  d,r
    host_notifications_enabled  1
}

```

The analogous directive for receiving service notifications is `service_notifications_enabled`.

8. Validate the configuration and restart the Nagios Core server:

```

# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload

```

With this done, the notifications sent when the host enters the `DOWN` state and also when it recovers into `OK` state should be sent to the contact. But no other notifications about the host (such as `UNREACHABLE` or `DOWNTIMESTART` notifications) should be sent about the `sparta.example.net` host to the contact `john`.

# How it works...

The kind of notifications a host should produce is defined in its `notification_options` directive and the kind of notifications that a defined contact should receive is configured in its `host_notification_options` directive. Both take the form of comma-separated values.

The valid values for the `notification_options` directive for host objects are:

- `d`: Notifications for a host being in the `DOWN` state
- `u`: Notifications for a host being in the `UNREACHABLE` state
- `r`: Notifications for a host recovering from problems (becoming `UP`)
- `f`: Notifications for a host beginning or ending the flapping state
- `s`: Notifications for a host beginning or ending its scheduled downtime

For example, we might define a host that should only send notifications when entering the `DOWN` or `UP` states, but ignore the notification events for `UNREACHABLE` or flapping, as follows:

```
define host {  
    ...  
    notification_options d,r  
}
```

For service objects, the valid directives are slightly different:

- `w`: Notifications for a service being in the `WARNING` state
- `u`: Notifications for a service being in the `UNKNOWN` state
- `c`: Notifications for a service being in the `CRITICAL` state
- `r`: Notifications for a service recovering from problems (becoming `OK`)
- `f`: Notifications for a service beginning or ending the flapping state
- `s`: Notifications for a service beginning or ending its scheduled downtime

For example, for a host that should send notifications for all of these events except for the flapping and scheduled downtime states, we might define:

```
define service {  
    ...  
    notification_options w,u,c,r  
}
```

All of the preceding values can be used in contact object definitions to restrict the service notifications that a particular contact should receive, through the `service_notification_options` directive:

```
define contact {  
    ...  
    service_notifications_enabled 1  
    service_notification_options w,u,c,r  
}
```

## There's more...

Unless we have a specific reason for particular contacts not to accept notifications of a particular kind, it's good practice to configure the contacts to receive all the notifications that they are sent, by including all the flags in both the directives for the contact:

```
define contact {  
    ...  
    host_notification_options    d,u,r,f,s  
    service_notification_options w,u,c,r,f,s  
}
```

This means that the contact will process any notification dispatched to it and we can instead restrict the kinds of notifications that are sent out by the hosts and services for which this is the delegated contact. With new configurations, it is generally better to start by sending too much information and then removing the unnecessary notification flags if appropriate, rather than potentially missing important messages because they're never sent.

If we intend to do this for all of our contacts, it may be appropriate to include these directives as part of a contact template and inherit from it using the `use` directive. The `generic-contact` contact template included with the default configuration may be suitable.

### Tip

If you want to completely disable the notifications for a given host, service, or contact, the single value `n` for `notification_options`, `host_notification_options` and `service_notification_options` will do this.



## See also

- The *Configuring notification for groups* section in this chapter
- The *Automating contact rotation* section in this chapter
- The *Defining a custom notification method* section in this chapter
- *Using inheritance to simplify configuration*, [Chapter 9](#), *Managing Configuration*



# Specifying the number of failed checks before notification

In this recipe, we'll learn how to arrange Nagios Core configuration to send notifications about problems with a host or service only after a check has been repeated a certain number of times and failed each time.

This can be an ideal arrangement for non-critical hosts that occasionally have “blips” or short outages and only become problematic if they remain down after repeated checks. It prevents sending notifications on the first check, only sending them if the problem turns out to be consistent.

# Getting ready

You should have a Nagios Core 4.0 or newer server, with at least one host configured already. We'll use the example of `sparta.example.net`, a host defined in its own file. We'll arrange for it to send us notifications only after a total of five failed host checks.

# How to do it...

We can configure the number of failed checks to be tolerated before sending a notification as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing your host definition and find the definition within it. It may look something like this:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    notification_period 24x7
}
```

3. Add or edit the value for `max_check_attempts` to 5:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    notification_period 24x7
    max_check_attempts 5
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, Nagios Core will not send a notification for the host entering the `DOWN` state until the check has been attempted a total of five times and has failed each time. In the following screenshot, note that even though four checks have already taken place and failed, no notification has been sent, and the state is listed as `SOFT`:

|                                     |                                             |
|-------------------------------------|---------------------------------------------|
| <b>Host Status:</b>                 | <b>DOWN</b> (for 0d 0h 0m 8s)               |
| <b>Status Information:</b>          | CRITICAL - Network Unreachable (192.0.2.21) |
| <b>Performance Data:</b>            |                                             |
| <b>Current Attempt:</b>             | 4/5 (SOFT state)                            |
| <b>Last Check Time:</b>             | 07-13-2015 22:31:40                         |
| <b>Check Type:</b>                  | ACTIVE                                      |
| <b>Check Latency / Duration:</b>    | 0.000 / 0.010 seconds                       |
| <b>Next Scheduled Active Check:</b> | 07-13-2015 22:32:40                         |
| <b>Last State Change:</b>           | 07-13-2015 22:31:40                         |
| <b>Last Notification:</b>           | N/A (notification 0)                        |
| <b>Is This Host Flapping?</b>       | <b>NO</b> (5.07% state change)              |
| <b>In Scheduled Downtime?</b>       | <b>NO</b>                                   |
| <b>Last Update:</b>                 | 07-13-2015 22:31:47 ( 0d 0h 0m 1s ago)      |

The SOFT state shows that while Nagios Core has flagged the host as DOWN, it will retry the check until it exhausts its `max_check_attempts`, at which point it will flag the host as being in a HARD DOWN state and send a notification.

However, if the host were to come back up before the next check, the state would change back to UP, without ever having sent a notification:

|                                     |                                                                |
|-------------------------------------|----------------------------------------------------------------|
| <b>Host Status:</b>                 | <b>UP</b> (for 0d 0h 0m 6s)                                    |
| <b>Status Information:</b>          | PING OK - Packet loss = 0%, RTA = 0.24 ms                      |
| <b>Performance Data:</b>            | rta=0.240000ms;3000.000000;5000.000000;0.000000 pl=0%;80;100;0 |
| <b>Current Attempt:</b>             | 1/5 (SOFT state)                                               |
| <b>Last Check Time:</b>             | 07-13-2015 22:32:18                                            |
| <b>Check Type:</b>                  | ACTIVE                                                         |
| <b>Check Latency / Duration:</b>    | 0.000 / 4.006 seconds                                          |
| <b>Next Scheduled Active Check:</b> | 07-13-2015 22:37:19                                            |
| <b>Last State Change:</b>           | 07-13-2015 22:32:22                                            |
| <b>Last Notification:</b>           | N/A (notification 0)                                           |
| <b>Is This Host Flapping?</b>       | <b>NO</b> (11.05% state change)                                |
| <b>In Scheduled Downtime?</b>       | <b>NO</b>                                                      |
| <b>Last Update:</b>                 | 07-13-2015 22:32:27 ( 0d 0h 0m 1s ago)                         |

## How it works...

The preceding configuration alters the `max_check_attempts` directive for the host to specify the number of checks that need to have failed before the host flags a `HARD DOWN` or `UNREACHABLE` state and generates a notification event.

The process for changing the maximum number of attempts before a notification for a service is identical; we add the same directive and value to the definition for the service:

```
define service {  
    use                generic-service  
    host_name          sparta.example.net  
    service_description HTTP  
    check_command       check_http  
    max_check_attempts 5  
}
```

## There's more...

The time between successive retry checks of a host before sending any notification can also be customized with the `retry_interval` directive. By default, the interval is in minutes, so if we wanted to configure a two-minute wait between the retry checks, we could add this directive to the host or service:

```
define host {  
    ...  
    retry_interval 2  
}  
define service {  
    ...  
    retry_interval 2  
}
```



## See also

- *Specifying how frequently to check a host or service, [Chapter 3](#), Working with Checks and States*
- *Managing brief outages with flapping, [Chapter 3](#), Working with Checks and States*



# Automating contact rotation

In this recipe, we'll learn how to automatically arrange notifications so that they are only received by certain contacts in a group at certain times.

For companies with more than one person as dedicated network staff, it's common practice to have an on-call roster, where one member of staff will be the dedicated on-call person for a certain period of time, perhaps a week or two. The kind of setup we'll configure here allows filtering notifications so that they're only delivered to the contacts at appropriate times; notifications are generated by hosts and services and sent to all the contacts in the nominated group, but only one (or perhaps more than one) of the contacts actually receives it.

We'll set this up using two properties of contacts that allow us to restrict the time period within which they should receive notifications: `host_notification_period` and `service_notification_period`.

# Getting ready

You should have a Nagios Core 4.0 or newer server, with at least one contact group configured already, and with at least two contacts within it. You should be familiar with configuring time periods and understand the basics of notifications as discussed earlier in this chapter.

We'll use the very simple example of three network operators, named as contacts `alan`, `brenden`, and `charlotte`, taking turns with monitoring duty every week. All three are members of the group `ops`, which is specified in the `contact_groups` directive for all the hosts and services on the network.

# How to do it...

We can set up a basic alternating schedule for receiving notifications for our operators as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Check the relevant file or files to ensure your contacts are defined and that they are members of the appropriate contact group. A good location for such information is in the `contacts.cfg` file.

```
define contactgroup {
    contactgroup_name ops
    alias             Network Operations
    members           alan,brenden,charlotte
}

define contact {
    use               generic-contact
    contact_name      alan
    alias             Alan Jones
    email             alan@pager.example.net
}

define contact {
    use               generic-contact
    contact_name      brenden
    alias             Brenden Peters
    email             brenden@pager.example.net
}

define contact {
    use               generic-contact
    contact_name      charlotte
    alias             Charlotte Franks
    email             charlotte@pager.example.net
}
```

3. In this case, all our hosts and services are configured to send messages to the ops contact group:

```
define host {
    ...
    contact_groups ops
}

define service {
    ...
    contact_groups ops
}
```

4. Define time periods to correspond to the times during which each of your contacts should be receiving messages. We'll use the special syntax of time period definitions to set up the definitions for this rotation:

```

define timeperiod {
    timeperiod_name    alan-pager
    alias               Alan pager schedule
    2015-12-05 / 21    00:00-24:00
    2015-12-06 / 21    00:00-24:00
    2015-12-07 / 21    00:00-24:00
    2015-12-08 / 21    00:00-24:00
    2015-12-09 / 21    00:00-24:00
    2015-12-10 / 21    00:00-24:00
    2015-12-11 / 21    00:00-24:00
}
define timeperiod {
    timeperiod_name    brenden-pager
    alias               Brenden pager schedule
    2015-12-12 / 21    00:00-24:00
    2015-12-13 / 21    00:00-24:00
    2015-12-14 / 21    00:00-24:00
    2015-12-15 / 21    00:00-24:00
    2015-12-16 / 21    00:00-24:00
    2015-12-17 / 21    00:00-24:00
    2015-12-18 / 21    00:00-24:00
}
define timeperiod {
    timeperiod_name    charlotte-pager
    alias               Charlotte pager schedule
    2015-12-19 / 21    00:00-24:00
    2015-12-20 / 21    00:00-24:00
    2015-12-21 / 21    00:00-24:00
    2015-12-22 / 21    00:00-24:00
    2015-12-23 / 21    00:00-24:00
    2015-12-24 / 21    00:00-24:00
    2015-12-25 / 21    00:00-24:00
}

```

5. Go back, and configure each of your contacts with the `host_notification_period` and `service_notification_period` directives to filter notifications to be received only during their dedicated time period:

```

define contact {
    use                       generic-contact
    contact_name              alan
    alias                     Alan Jones
    email                     alan@pager.example.net
    host_notification_period    alan-pager
    service_notification_period alan-pager
}
define contact {
    use                       generic-contact
    contact_name              brenden
    alias                     Brenden Peters
    email                     brenden@pager.example.net
    host_notification_period    brenden-pager
    service_notification_period brenden-pager
}
define contact {
    use                       generic-contact

```

```
    contact_name          charlotte
    alias                 Charlotte Franks
    email                 charlotte@pager.example.net
    host_notification_period  charlotte-pager
    service_notification_period charlotte-pager
}
```

6. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, the contacts configured should receive notifications only during their dedicated pager time. If these are our only contacts, then all the notifications generated will go to only one person at any given time.

# How it works...

The `host_notification_period` and `service_notification_period` directives in the definitions for our contacts are used to limit the times during which these contacts should receive any notifications. Each of our contacts in this example has their own time period defined to correspond to their portion of the pager schedule.

Let's take a look at Alan's example:

```
define timeperiod {
    timeperiod_name    alan-pager
    2015-12-05 / 21    00:00-24:00
    2015-12-06 / 21    00:00-24:00
    2015-12-07 / 21    00:00-24:00
    2015-12-08 / 21    00:00-24:00
    2015-12-09 / 21    00:00-24:00
    2015-12-10 / 21    00:00-24:00
    2015-12-11 / 21    00:00-24:00
}
```

The second line of the directive, `2015-12-05 / 21 00:00-24:00`, can be broken down as follows:

- Starting from the 5th of December
- Every 21 days,
- From midnight to the following midnight (that is, the entire day)

The configuration then goes on to do the same for the remaining days of the first week when Alan will be on call and specifies that he will be on call the same day 21 days (three weeks) later. A similar configuration, but starting one week and two weeks later, is set up for the contacts `brenden` and `charlotte` respectively.



## There's more...

Note that there's no reason time periods can't overlap. If it happens to suit us to arrange for more than one contact to receive a notification, we can do that. Similarly, we can leave gaps in the schedule if notifications aren't needed during a certain time period.

Time periods in Nagios Core are highly flexible; to see some examples of the various syntaxes you can use for defining them, take a look at the "Creating a new time period" recipe in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

## See also

- *Creating a new contact, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new contact group, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The [Configuring notifications for groups](#) section in this chapter*
- *Creating a new time period, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The [Choosing states for notification](#) section in this chapter*



# Defining an escalation for repeated notifications

In this recipe, we'll learn how to arrange a Nagios Core configuration such that after a certain number of repetitions, notifications for problems on hosts or services are escalated to another contact, instead of (or in addition to) the normally defined contact. This is done by defining a separate object type called a **host** or **service escalation**.

This kind of setup could be useful for alerting more senior networking staff of an unsolved problem that a less experienced person is struggling to fix and can also function as a “safety valve” to ensure that problem notifications for hosts eventually do reach someone else if they remain unfixed.

# Getting ready

You should have a Nagios Core 4.0 or newer server, with at least one host or service configured already, and at least two contact groups — one for the first few notifications, and one for the escalations. You should understand how notifications are generated and sent to the `contacts` and `contact_groups` for hosts or services.

We'll use the example of a host called `sparta.example.net`, which normally sends notifications to a group called `ops`. We'll arrange for all the notifications after the fourth one to also be sent to a contact group called `emergency`.

# How to do it...

We can configure an escalation for our host or service as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the host. The definition might look like this:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta
    address            192.0.2.21
    contact_groups     ops
    notification_period 24x7
    notification_interval 10
}
```

3. Underneath the host definition, add the configuration for a new hostescalation object:

```
define hostescalation {
    host_name          sparta.example.net
    contact_groups     ops,emergency
    first_notification 5
    last_notification  0
    notification_interval 10
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, when problems are encountered with the host that generate notifications, all the notifications beyond the fourth one will be sent to both the ops contact group and also to the emergency contact group, expanding the number of people or systems contacted, and making it more likely that the problem is actually addressed or fixed. Perhaps someone on the ops team has misplaced their pager.

## How it works...

The configuration added in the preceding section is best thought of as a special case or override for a particular host, in that it specifies a range of notifications that should be sent to a different set of contact groups. It can be broken down as follows:

- `host_name`: The same value of `host_name` given for the host in its definition. We specified `sparta.example.net`.
- `contact_groups`: The contact groups to which the notifications meeting this special case should be sent. Note that we specify both the emergency group and the ops group, so that the matching notifications go to both. Note that they are comma-separated.
- `first_notification`: The count of the first notification that should match this escalation. We chose the fifth notification.
- `last_notification`: The count of the last notification that should match this escalation. We set this to zero, which means that all notifications after the `first_notification` should be sent to the nominated contact or contact groups. The notifications will not stop until they are manually turned off, or the problem is fixed.
- `notification_interval`: Like the host and service directives of the same name, this specifies how long Nagios Core should wait before sending new notifications if the host remains in a problematic state. Here, we've chosen ten minutes.

Individual contacts can also (or instead) be specified with the `contacts` directive, rather than `contact_groups`.

## There's more...

The preceding escalation continues sending notifications both to the original ops group and also to the members of the emergency group. It's generally a good idea to do it this way rather than sending notifications only to the escalated group, because the point of escalations is to increase the reach of the notifications when a problem is not being dealt with, rather than merely trying to contact a different group of people instead.

This principle applies to stacking escalations as well; if we had a host group with all our contacts in it, perhaps named everyone, we could define a second escalation that from the tenth notification onwards goes to every single contact:

```
define hostescalation {
    host_name           sparta.example.net
    contact_groups       everyone
    first_notification   10
    last_notification    0
    notification_interval 10
}
```

Just as we can specify multiple host escalations, it's also fine for the ranges of notifications to overlap so that more than one escalation applies.

With a little arithmetic, you can arrange escalations such that they work after a host or service has been in a problematic state for a certain period of time. For example, the escalation we specified in the recipe will apply after the host has been in a problematic state for 40 minutes, because the `notification_interval` specifies that Nagios Core should wait 10 minutes between resending notifications.

Service escalations work much the same way as host escalations do; the difference is that you need to specify the service by its `service_description` as well as its host name. Everything else works the same way. An escalation for a service check called HTTP running on `sparta.example.net` that does the same thing as the previous escalation would look like this:

```
define serviceescalation {
    host_name           sparta.example.net
    service_description HTTP
    contact_groups       ops,emergency
    first_notification   5
    last_notification    0
    notification_interval 10
}
```



## See also

- *Creating a new contact group*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- The *Configuring notifications for groups* section in this chapter
- The *Defining a custom notification method* section in this chapter



# Defining a custom notification method

In this recipe, we'll learn how to specify an alternative method for a contact to receive notifications about a service. A very typical method for a contact to receive notifications is by being sent an e-mail to their contact address; e-mail messages could be sent to an inbox or a paging device.

However, notifications are just text; we can arrange to deal with them via any command we wish, in much the same way we can configure host or service checks. In this recipe, we'll set up a new contact called `motd`, which, when it receives notifications, will write them into the server's `/etc/motd` to be displayed on login.

# Getting ready

You should have a Nagios Core 4.0 or newer server, with at least one host or service configured already. You should understand how notifications are generated and their default behavior in being sent to the contacts and contact\_groups for hosts or services.

We'll use the example of a host called `troy.example.net`, configured to send notifications to an ops contact group. We'll add a new contact within this group called `motd`.

# How to do it...

We can arrange our custom notification method as follows:

1. Ensure that your `motd(5)` file can be written to by Nagios Core and that it's a static file that your system will not overwrite on boot. If you run your Nagios Core server as the recommended default user and group of `nagios`, you could do something like this:

```
# chgrp nagios /etc/motd
# chmod g+w /etc/motd
```

2. It would be a good idea to test that the user can write to the file using `su(8)` or `sudo(8)`:

```
# sudo -s -u nagios
$ printf 'test\n' >>/etc/motd
```

3. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

4. Edit the file containing the definitions for your notification commands. By default, this is in the file `commands.cfg`. Append the following definitions to the file:

```
define command {
    command_name    notify-host-motd
    command_line    /usr/bin/printf "%s\n" "$SHORTDATETIME$:
$HOSTALIAS$ is $HOSTSTATE$" >>/etc/motd
}
define command {
    command_name    notify-service-motd
    command_line    /usr/bin/printf "%s\n" "$SHORTDATETIME$:
$SERVICEDESC$ on $HOSTALIAS$ is $SERVICESTATE$" >>/etc/motd
}
```

5. Edit the file containing the definitions for your contacts. By default, this is in the file `contacts.cfg`. Append the following definitions to the file:

```
define contact {
    use                generic-contact
    contact_name        motd
    contact_groups      ops
    alias               MOTD
    host_notification_commands    notify-host-motd
    service_notification_commands    notify-service-motd
}
```

6. Edit the file containing the definitions for your host and/or services to ensure that they specify `ops` in their `contact_groups` directives:

```
define host {
    host_name          troy.example.net
```

```
    ...  
    contact_groups ops  
}  
define service {  
    host_name      troy.example.net  
    ...  
    contact_groups ops  
}
```

7. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, the next notifications that go out for the host and services should not only be e-mailed to any contacts in the ops group, but should also be written to the MOTD to be presented on login:

```
06-30-2012 17:24:05: troy is DOWN  
06-30-2012 17:24:05: HTTP on troy is CRITICAL  
06-30-2012 17:25:07: troy is DOWN  
06-30-2012 17:25:07: HTTP on troy is CRITICAL
```

# How it works...

The first part of the configuration that we added is defining new notification commands. These are actually command definitions, just the same as those used with the `check_command` definitions for hosts and services; the difference is that they're used by contacts to send notifications to people (or in this case, write them to files) in the appropriate way.

If you're using the default Nagios Core configuration, the commands already defined in `commands.cfg` should include `notify-host-by-email` and `notify-service-by-email`. I've truncated the complete command line in the following code as it's very long:

```
define command {
    command_name      notify-host-by-email
    command_line       /usr/bin/printf "%b" "***** Nagios
*****\n\nNotification Type: $NOTIFICATIONTYPE$\nHost: $HOSTNAME$\nState:
$HOSTSTATE$\nAddress: $HOSTADDRESS$\nInfo: $HOSTOUTPUT$\n\nDate/Time:
$LONGDATETIME$\n" | /usr/bin/mail -s "*** $NOTIFICATIONTYPE$ Host Alert:
$HOSTNAME$ is $HOSTSTATE$ ***" $CONTACTEMAIL$
}
define command {
    command_name      notify-service-by-email
    command_line       /usr/bin/printf "%b" "***** Nagios
*****\n\nNotification Type: $NOTIFICATIONTYPE$\n\nService:
$SERVICEDESC$\nHost: $HOSTALIAS$\nAddress: $HOSTADDRESS$\nState:
$SERVICESTATE$\n\nDate/Time: $LONGDATETIME$\n\nAdditional
Info:\n\n$SERVICEOUTPUT$\n" | /usr/bin/mail -s "*** $NOTIFICATIONTYPE$
Service Alert: $HOSTALIAS$/$SERVICEDESC$ is $SERVICESTATE$ ***"
$CONTACTEMAIL$
}
```

These commands use the system `printf` implementation, along with many Nagios Core macros, to build an appropriate notification string, which is e-mailed to the appropriate contact using a shell pipe into the system mailer.

If we inspect the `templates.cfg` file, where the generic-contact template is defined, we can see these methods are referred to in two directives:

```
define contact {
    ...
    service_notification_commands notify-service-by-email
    host_notification_commands    notify-host-by-email
}
```

This configuration thus defines a particular notification command to be used for hosts that derive from the generic-host template.

Our own command definition does something similar in building a string with a call to `printf`; but instead of writing the output into a pipe to the system mailer, it appends it to the `/etc/motd` file.

The `command_line` we specified for the `motd` contact to process host notifications is as follows:

```
/usr/bin/printf "%s\n" "$SHORTDATETIME$: $HOSTALIASE$ is $HOSTSTATE$"  
>>/etc/motd
```

The first thing Nagios Core does when a notification event for this contact is successfully fired is substitute values for its macros, \$SHORTDATETIME\$, \$HOSTALIASE\$, and \$HOSTSTATE\$:

```
/usr/bin/printf "%s\n" "06-30-2012 17:24:05: troy is DOWN" >>/etc/motd
```

The resulting string is then executed as a command by the running Nagios Core user, by default nagios, and provided that the user has the appropriate permissions, it's appended to the end of the MOTD.



## There's more...

Adding messages to the MOTD may not be useful for all network administrators, particularly for larger or more sensitive networks for which hundreds of notifications could be generated daily. The main thing to take away from this is that Nagios Core can be configured to process notification events in pretty much any way, provided the following conditions are met:

- We can define a command line that will consistently do what is needed with the results of the macro substitution Nagios Core will perform for us
- The nagios user, or whichever user the Nagios Core server is running, has all the relevant permissions it needs to run the command

E-mail notifications, particularly to a mobile paging device, are a very sensible basis for notifications for most administrators. However, if some other means of notification is needed, such as inserting into a database or piping into a Perl script, all that is possible as well.

Note that it's safest to specify the full paths to any binaries or files we use in these commands, hence `/usr/bin/printf` rather than merely `printf`. This is to ensure that the program we actually intend to run is run and avoid having to mess about with `$PATH` specifications for the nagios user.

## See also

- *Creating a new command*, [Chapter 2](#), *Working with Commands and Plugins*
- *Customizing an existing command*, [Chapter 2](#), *Working with Commands and Plugins*
- The *Configuring notifications for groups* section in this chapter



# Filtering notifications based on a host or service value

One of the new features of Nagios Core 4.0 is a means for specifying a value for hosts and services to allow for setting a threshold for notification behavior. This is implemented with the `importance` directive for hosts and services and the `minimum_importance` directive for contacts. This feature is designed to allow some flexibility for deciding how critical problems with specific hosts or services are and, hence, whether to alert different contacts depending on their severity.

For example, it might be appropriate to page a system administrator about a server used for internal development being inaccessible, but not to contact anybody else, as this service being down doesn't involve accountability to the public, or a great deal of lost revenue. For such a service, we would set a low value for the `importance` directive. However, for problems with a service such as a public-facing website for a government organization, it may be appropriate to notify both system administrators and management about problems and so we would set a high value for the `importance` directive.

For this example, we'll define a service on one web server corresponding to a development website and one on another corresponding to a public website. We'll then apply the `importance` and `minimum_importance` directives to filter notifications appropriately to the system administration and management teams.

# Getting ready

You should have a Nagios Core 4.0.3 or newer server, with at least one host or service configured already. You should understand how notifications are generated and their default behavior in being sent to the contacts and contact\_groups for hosts or services.

We'll use the example of a host called `sparta.example.net`, hosting a relatively unimportant development website, and one called `athens.example.net`, hosting a critical public-facing website. Both hosts are configured to send notifications to adam, a systems administrator, and brenden, a manager.

# How to do it...

We can apply an importance for our hosts and a `minimum_importance` for our contacts as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file or files containing your host definitions. For each relevant host, set the `importance` directive to some suitable value. Ours might look like the following; we set a much higher importance for the more critical host:

```
define host {
    use                linux-server
    host_name          sparta.example.net
    alias              sparta (development webserver)
    address            192.0.2.21
    notification_period 24x7
    contacts           adam,brenden
    importance         15
}
define host {
    use                linux-server
    host_name          athens.example.net
    alias              athens (production webserver)
    address            192.0.2.22
    notification_period 24x7
    contacts           adam,brenden
    importance         100
}
```

3. Edit the file or files containing your contact objects, and set the `minimum_importance` for each to specify how urgent a notification needs to be before it is sent to that contact:

```
define contact {
    use                generic-contact
    contact_name       adam
    alias              Adam (Sysadmin)
    email              sysadmin@example.net
    minimum_importance 10
}
define contact {
    use                generic-contact
    contact_name       brenden
    alias              Brenden (Manager)
    email              manager@example.net
    minimum_importance 50
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, if the `sparta.example.net` server in this example were to go DOWN or become UNREACHABLE, only adam would receive notifications, even though brenden is listed as a contact. However, if `athens.example.net` goes down, both adam and brenden receive notifications.

## How it works...

The importance for `sparta.example.net` was set at 15. When Nagios Core notices that it's DOWN, after completing all its retries, it triggers a notification. Looking at the `contacts` directive, it finds that `adam` and `brenden` are both set as contacts for the host. It notes that the importance for the host is greater than zero and compares it to the `minimum_importance` for each contact. Because 15 is greater than `adam`'s `minimum_importance` value of 10, a notification is sent to him, but, because it's less than `brenden`'s `minimum_importance` of 50, no notification is sent to `brenden`.

However, the importance value of `athens.example.net` is much higher, at 100. When it goes DOWN, notifications are sent to both `adam` and `brenden`, because the importance of 100 is greater than both of their `minimum_importance` settings.

This method only works if both the host's importance value and the contact's `minimum_importance` value are greater than zero, the default.



## There's more...

In practice, for this simple example, it's likely that we'd instead simply specify only adam as the contact for sparta.example.net, which would have the same effect as our previous recipe. However, for large and complex setups with a lot of hosts, services, and contacts, using a numeric scale for gauging whether notifications should be sent to contacts may be more flexible than managing associated contacts per host or service. This is especially likely to be the case if your Nagios Core configuration is large and generated from some other tool, such as a configuration management database.

Adding to this flexibility, if you define importance for a host's services as well as for the host itself, then the importance for the host is calculated as the host value and all of its service values, added together.

From Nagios Core 4.0.0 through 4.0.2, the directives were named `hourly_value` and `minimum_value` instead of `importance` and `minimum_importance`; they were renamed in Nagios Core 4.0.3 and the old values are now deprecated. If you're using a version of Nagios Core 4.0.3 or beyond, you should use the new names instead to avoid warnings.

## See also

- The *Choosing states for notification* section in this chapter
- The *Defining an escalation for repeated notifications* section in this chapter



# Chapter 5. Monitoring Methods

In this chapter, we will cover the following topics:

- Monitoring PING for any host
- Monitoring SSH for any host
- Checking an alternative SSH port
- Monitoring mail services
- Monitoring web services
- Checking that a website returns a given string
- Monitoring database services
- Monitoring the output of an SNMP query
- Monitoring a RAID or other hardware device
- Creating an SNMP OID for monitoring

# Introduction

Nagios Core is best thought of as a monitoring framework that uses plugins to perform appropriate checks on hosts and services, and returns results about their states in a format that it understands and can use for sending notifications and keeping track of states on a longer term basis.

The design is quite flexible; as explained in the “Commands and Plugins” chapter, Nagios Core can use as a plugin any command-line application that gives appropriate return values as defined in the Nagios Core header files, Perl library, or shell script. In turn, Nagios Core can be configured to use the same plugin in many different ways, taking advantage of any switches provided by the plugin to adjust its behavior, including providing metadata to it in the form of the values of Nagios Core **macros** like `$HOSTADDRESS$`.

The collection of plugins available on the Nagios Exchange website at <https://exchange.nagios.org/> is fairly large and documenting all of them is well out of the scope of this book. However, some of the most useful plugins are included as part of the Nagios Plugins set and are installed as part of the recommended quick start guides for Nagios Core at <https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/>. They include programs for monitoring very common network features in typical ways, such as monitoring basic network connectivity, web services, mail servers, and other very common network properties and services. The plugin site itself is at <https://nagios-plugins.org/>.

This chapter will demonstrate the usage of some of the most useful components of this plugin set, which it assumes you have already installed. The focus will be on monitoring tasks that will be relevant to most or even all networks of various sizes, hopefully bringing the reader well past the point of thinking of Nagios Core as merely a process for sending PING requests. The last few recipes will show how you can use the **Simple Network Management Protocol (SNMP)** as a method of checking any generic network service or system property that may not be covered by the standard Nagios Plugins set.



# Monitoring PING for any host

In this recipe, we'll learn how to set up PING monitoring for a host. We'll use the `check_ping` plugin, and its command of the same name, to send ICMP ECHO requests to a host. We'll use this as a simple diagnostic check to make sure that the host's network stack is responding in a consistent and timely fashion, in much the same way an administrator might use the `ping(8)` command interactively to check the same properties.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `corinth.example.net`, a host defined in its own file. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.



# How to do it...

We can add a new PING service check to our existing host as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {
    use                windows-server
    host_name          corinth.example.net
    alias              corinth
    address            192.0.2.27
}
```

3. Beneath the definition for the host, place a service definition referring to `check_ping`. It's recommended that you use the generic-service template, as follows:

```
define service {
    use                generic-service
    host_name          corinth.example.net
    service_description PING
    check_command       check_ping!100,20%!200,40%
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, a new service check will start taking place, with the appropriate contacts and contact groups notified in the following scenarios:

- The RTT (Round Trip Time) of the request and its response exceeds 200 ms, OR more than 40% of the packets are lost during the check; a CRITICAL notification is fired for the service in either case.
- If a CRITICAL notification was not fired, and the RTT of the request and its response exceeds 100 ms, OR more than 20% of the packets are lost during the check; in this case, a WARNING notification is fired for the service.

The information about the thresholds is given in the definition for the `check_command` directive as arguments to the `check_ping` command.

Information about this service will also be visible in the web interface under the **Services** section.

## How it works...

The preceding configuration defines a new service check on the existing `corinth.example.net` host to check that the RTT and the packet loss for an ICMP ECHO request and response are within acceptable limits.

For most network configurations, it may well be the case that the host itself is also being checked by `check_ping` by way of the command `check-host-alive`. The difference is that the thresholds for RTT and packet loss are intentionally set very high for this command, because it is intended to ensure whether the host is up or down at all, not how responsive it is.

## There's more...

In networks where all the hosts are configured to respond to ICMP ECHO requests, it could perhaps be worthwhile to configure service checks on all of them in a configuration. This can be done using the `*` wildcard when defining the `host_name` for the service:

```
define service {
    use                generic-service
    host_name          *
    service_description PING
    check_command       check_ping!100,20%!200,40%
}
```

This will apply the same check, with a `service_description` of `PING`, to all the hosts configured in the database. This method will save the hassle of configuring a service separately for all the hosts.

If some of the hosts in a network do not respond to `PING`, it may be more appropriate to place the ones that do in a **hostgroup**, perhaps named something like `icmp`:

```
define hostgroup {
    hostgroup_name icmp
    alias          ICMP enabled hosts
    members        sparta.example.net,corinth.example.net
}
```

The single service can then be applied to all the hosts in that group using the `hostgroup_name` directive in the service definition:

```
define service {
    use                generic-service
    hostgroup_name     icmp
    service_description PING
    check_command       check_ping!100,20%!200,40%
}
```

It's generally a good idea to have network hosts respond to ICMP messages where possible in order to comply with the recommendations in RFC1122 and to ease debugging.

Finally, note that the thresholds for the round trip time and packet loss are not fixed; in fact, they're defined in the service definition, in the `check_command` line. For hosts that have higher latency, perhaps due to network load or topology, it may be appropriate to adjust these thresholds; this is covered in the recipe *Changing thresholds for ping RTT and packet loss* in [Chapter 3, Working with Checks and States](#).

## See also

- *Creating a new host, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new hostgroup, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Changing thresholds for ping RTT and packet loss, [Chapter 3](#), Working with Checks and States*
- *Using an alternative check command, [Chapter 2](#), Working with Commands and Plugins*



# Monitoring SSH for any host

In this recipe, we'll learn how to check that the SSH daemon on a remote host is responding to requests using the `check_ssh` plugin and the command of the same name. This will allow us to be notified as soon as there are problems connecting to the SSH service.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `troy.example.net`, a host defined in its own file. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

It may be a good idea to verify first that the host for which you want to add monitoring is presently running the SSH service that requires checking. This can be done by running the `ssh(1)` client to make a connection to the host:

```
$ ssh troy.example.net
```

We should also check that the plugin itself will return the result required when run against the applicable host as the nagios user:

```
# sudo -s -u nagios  
$ /usr/local/nagios/libexec/check_ssh troy.example.net
```

If you're unable to get a positive response from the SSH service on the target machine even if you're sure it's running, this could perhaps be a symptom of unrelated connectivity or filtering problems. We may, for example, need to add the monitoring server on which Nagios Core is running to the whitelist for SSH (normally TCP destination port 22) on any applicable firewalls or routers.

# How to do it...

We can add a new SSH service check to our existing host as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {
    use          linux-server
    host_name    troy.example.net
    alias        troy
    address      192.0.2.25
}
```

3. Beneath the definition for the host, place a service definition referring to `check_ssh`. It may help to use the `generic-service` template or another suitable template, as follows:

```
define service {
    use          generic-service
    host_name    troy.example.net
    service_description SSH
    check_command check_ssh
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, a new service check will start taking place with the appropriate contacts and contact groups being notified when an attempt to connect to the SSH server fails. The service check will be visible in the web interface on the **Services** page.



## How it works...

The preceding configuration defines a new service with a `service_description` of SSH for the existing `troy.example.net` host using the values in the `generic-service` template, and additionally defining a `check_command` of `check_ssh`.

This means that other than checking if the host itself is up with `check-host-alive`, as was being done previously, Nagios Core will also check that the SSH service running on the host is working by attempting to make a connection to it; this includes notifying the applicable contacts if there are any problems found with the service after the appropriate number of tests.

For example, if the plugin finds that the host is accessible but not responding to client tests, it might notify with the following text:

```
Subject: ** PROBLEM Service Alert: troy.example.net/SSH is CRITICAL **
***** Nagios *****
Notification Type: PROBLEM
Service: SSH
Host: troy.example.net
Address: troy.example.net
State: CRITICAL
Date/Time: Mon Aug 27 21:15:12 NZST 2015
Additional Info:
CRITICAL - Socket timeout after 10 seconds
```

Note that we don't need to actually supply credentials for the SSH check; the plugin simply ensures that the service is running and responding to connection attempts.

## There's more...

The definition for the `check_ssh` command warrants some inspection if we're curious as to how the plugin is actually applied as a command, as defined (by default) in `/usr/local/nagios/etc/objects/commands.cfg`:

```
define command {  
    command_name    check_ssh  
    command_line    $USER1$/check_ssh $ARG1$ $HOSTADDRESS$  
}
```

This shows that the `check_ssh` command is configured to run the `check_ssh` binary file in `$USER1$`, a macro that normally expands to `/usr/local/nagios/libexec`, against the host address of the applicable server. It adds in any other arguments beforehand by expanding `$ARG1$`. We haven't used that macro in this recipe since we simply want to make a normal check of the SSH service on its default port.

This check should work with most SSH2 compliant servers, most notably including the popular OpenSSH server.

Checking SSH accessibility is a common enough thing for servers that it may well be worth setting up an SSH check service to apply for a **hostgroup**, rather than merely for an individual host. For example, if you had a group called `ssh-servers` containing several servers that should be checked with a `check_ssh` call, you could configure them all to be checked with one service definition using the `hostgroup_name` directive:

```
define service {  
    use                generic-service  
    hostgroup_name     ssh-servers  
    service_description SSH  
    check_command      check_ssh  
}
```

This would apply the same service check to each host in the group; this makes the definition easier to update if the check needs to be changed or removed in the future.

Note that the `check_ssh` plugin is different from the `check_by_ssh` plugin, which is used to run checks on remote machines, much like NRPE.

## See also

- *Creating a new host, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new hostgroup, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The `Checking an alternative SSH port` recipe in this chapter*
- *Using `check_by_ssh` with key authentication instead of NRPE, [Chapter 6](#), Enabling Remote Execution*
- *Monitoring local services on a remote machine with NRPE, [Chapter 6](#), Enabling Remote Execution*



# Checking an alternative SSH port

In this recipe, we'll learn how to deal with the common situation of a machine running an SSH daemon that is running on an alternative port; in this case, a service definition that uses `check_ssh` as used in the recipe *Monitoring SSH for any host* fails, because the plugin defaults to using the standard SSH TCP port number 22.

This kind of setup is common in situations where an SSH server should not be open to the general public and is often employed as a “security by obscurity” method to reduce automated attacks against the server. The SSH daemon is, therefore, configured to listen on a different port, usually with a much higher number, and the administrators who need to use it are told what the port number is.

We'll deal with this situation and monitor the service in Nagios Core, even though it's running on a non-standard port. We'll do this by defining a new command that checks SSH on a specified port number and creating a service definition that uses that command. The command will accept the port number to check as an argument.

The principles here should generalize well to any other situation where checking an alternative port is necessary and the Nagios Core plugin being used to make the check supports doing so on an alternative port.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `troy.example.net`, a host defined in its own file and listening on the non-standard SSH port 5022. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

A good first step may be to verify that we're able to access the SSH daemon from the monitoring server on the specified port. We can do this from the command line using the `ssh` client, specifying the port number with the `-p` option:

```
$ ssh troy.example.net -p 5022
```

Alternatively, you can run the `check_ssh` plugin directly from the command line:

```
# sudo -s -u nagios  
$ /usr/local/nagios/libexec/check_ssh -p 5022 troy.example.net
```

# How to do it...

We can set up a service check for SSH on a non-standard port as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit a suitable file containing command definitions and find the definition for the `check_ssh` command. In the default installation, this file is `commands.cfg`. The `check_ssh` definition looks something like this:

```
define command {  
    command_name    check_ssh  
    command_line    $USER1$/check_ssh $ARG1$ $HOSTADDRESS$  
}
```

3. Beneath the `check_ssh` definition, add a new command definition as follows:

```
define command {  
    command_name    check_ssh_altport  
    command_line    $USER1$/check_ssh -p $ARG1$ $HOSTADDRESS$  
}
```

4. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {  
    use              linux-server  
    host_name        troy.example.net  
    alias            troy  
    address          192.0.2.25  
}
```

5. Beneath the definition for the host, place a new service definition using our new command:

```
define service {  
    use                  generic-service  
    host_name            troy.example.net  
    service_description  SSH_5022  
    check_command         check_ssh_altport!5022  
}
```

6. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, Nagios Core will begin running service checks using the `check_ssh` plugin, but will use the alternative destination port 5022 for its connection attempts for the service with the description `SSH_5022`.

## How it works...

The preceding configuration has almost exactly the same end result as adding a default `check_ssh` service; the only difference is that a different port is checked in order to make the connection. We use the command `check_ssh_altport` to do this, which we also defined ourselves in a syntax very similar to the `check_ssh` definition.

The difference is that the command accepts an argument, which is used as a value for the `-p` option to the `check_ssh` plugin, to check the specified port number—in this case, TCP port 5022, rather than the default port 22.



## There's more...

Because arguments in Nagios Core can include spaces, we could also have defined the service check as follows, without having to define an extra command:

```
define service {  
    use                generic-service  
    host_name          troy.example.net  
    service_description SSH_5022  
    check_command       check_ssh! -p 5022  
}
```

This is because the `$ARG1$` macro representing the argument is still used in the original `check_ssh` command, but it needs to have the option as well as its value included. The difference is mainly one of preference, depending on which we feel is clearer and more maintainable. It may help to consider whether a well-named command could assist someone else reading our configuration to understand what is meant.

## See also

- *Creating a new host, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new command, [Chapter 2](#), Working with Commands and Plugins*
- *The Monitoring SSH for any host recipe in this chapter*



# Monitoring mail services

In this recipe, we'll learn how to monitor three common mail services for a nominated host: SMTP, POP, and IMAP. We'll also see how to use the same structure to include additional checks for secure, encrypted versions of each of these services: SMTPS, POPS, and IMAPS.

For simplicity, we'll assume in this recipe that all three of these services are running on the same host, but the procedure will generalize easily for the common case where there are designated servers for one or more of the previously mentioned functions.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `troy.example.net`, a host defined in its own file. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

Checking the connectivity for the required services on the target server is also a good idea to ensure that the automated connections the monitoring server will be making on the appropriate protocols and ports will actually work as expected. For the plain unencrypted mail services, this could be done via Telnet to the appropriate ports. For SMTP:

```
$ telnet marathon.example.net 25
Trying 192.0.2.34...
Connected to marathon.example.net.
Escape character is '^]'.
220 marathon.example.net ESMTP Postfix
```

For POP:

```
$ telnet marathon.example.net 110
Trying 192.0.2.34...
Connected to marathon.example.net.
Escape character is '^]'.
+OK Hello there.
```

And for IMAP:

```
$ telnet marathon.example.net 143
Trying 192.0.2.34...
Connected to marathon.example.net.
Escape character is '^]'.
* OK [CAPABILITY IMAP4rev1 UIDPLUS CHILDREN NAMESPACE...
```

For secure services, one possibility for checking is by using the `openssl(1)` client. For SMTPS on its classic port number 465:

```
$ openssl s_client -connect marathon.example.net:465
CONNECTED(00000003)
...
220 marathon.example.net ESMTP Postfix
```

For POPS:

```
$ openssl s_client -connect marathon.example.net:995
CONNECTED(00000003)
...
+OK Hello there.
```

And for IMAPS:

```
$ openssl s_client -connect marathon.example.net:993
CONNECTED(00000003)
...
* OK [CAPABILITY IMAP4rev1 UIDPLUS CHILDREN NAMESPACE...
```

If you prefer, you could instead use a network scanner such as Nmap to test if the ports are open and responsive.

Once we've verified connectivity for the mail services that we need and the host itself is being configured and checked in Nagios Core, then we can add the appropriate service checks.

# How to do it...

We can add unencrypted mail service checks for SMTP, POP, and IMAP services on our host as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {
    use          linux-server
    host_name    marathon.example.net
    alias        marathon
    address      192.0.2.34
}
```

3. Beneath the definition for the host, place three new service definitions, one each for the appropriate mail services:

```
define service {
    use          generic-service
    host_name    marathon.example.net
    service_description SMTP
    check_command check_smtp
}
define service {
    use          generic-service
    host_name    marathon.example.net
    service_description POP
    check_command check_pop
}
define service {
    use          generic-service
    host_name    marathon.example.net
    service_description IMAP
    check_command check_imap
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, three new service checks will start taking place, with the appropriate contacts and contact groups being notified when an attempt to connect to any of the services fails. Details for these services will also become available in the **Services** section of the web interface.

## How it works...

The preceding configuration adds three new service checks to the existing `marathon.example.net` host:

- SMTP, which uses the `check_smtp` command to open an SMTP session
- POP, which uses the `check_pop` command to open a POP session
- IMAP, which uses the `check_imap` command to open an IMAP session

In all three cases, the connectivity and responsiveness of the service is checked and determined to be OK if it returns appropriate values within an acceptable time frame.

It's important to note that this configuration that we just defined doesn't actually send or receive any e-mail messages; it merely checks the basic connectivity of the service and whether it answers simple requests. Therefore, just because the status is OK, it does not necessarily mean that e-mail messages are being delivered correctly—it just means that the services are responding.



## There's more...

If it's necessary to check the secure SSL/TLS versions of each of these services, the configuration is very similar, but it requires a little extra setup beforehand. This is because, although plugins to check them are included in the Nagios Plugins setup, they are not configured to be used as commands. Note that this may well change in future versions of Nagios Core.

To add the appropriate commands, the following stanzas could be added to the commands configuration file, normally `/usr/local/nagios/etc/commands.cfg`:

```
define command {
    command_name    check_ssmtp
    command_line    $USER1$/check_ssmtp $ARG1$ -H $HOSTADDRESS$
}
define command {
    command_name    check_spop
    command_line    $USER1$/check_spop $ARG1$ -H $HOSTADDRESS$
}
define command {
    command_name    check_simap
    command_line    $USER1$/check_simap $ARG1$ -H $HOSTADDRESS$
}
```

With this done, the following service definitions can then be added to the appropriate host, either replacing or supplementing the checks for the unsecured services. They are just the same as the unsecured versions, except an `s` is added to the `service_description` and to the `check_command`:

```
define service {
    use                generic-service
    host_name          marathon.example.net
    service_description SSMTP
    check_command       check_ssmtp
}
define service {
    use                generic-service
    host_name          marathon.example.net
    service_description SPOP
    check_command       check_spop
}
define service {
    use                generic-service
    host_name          marathon.example.net
    service_description SIMAP
    check_command       check_simap
}
```

Finally, note that if you are managing more than one mail server running one or more of these services, it's good practice to apply the service to a **hostgroup** containing all the applicable hosts, rather than creating new service definitions for each one. See the recipe *Running a service on all hosts in a group* in [Chapter 1, Understanding Hosts, Services, and Contacts](#), to learn how to do this.

## See also

- *Creating a new host, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new command, [Chapter 2](#), Working with Commands and Plugins*
- *Creating a new hostgroup, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Running a service on all hosts in a group, [Chapter 1](#), Understanding Hosts, Services, and Contacts*



# Monitoring web services

In this recipe, we'll set up a service check to monitor the responsiveness of an HTTP and HTTPS server. We'll use the `check_http` command and the plugin of the same name provided in the Nagios Plugins setup for making HTTP and HTTPS requests of a web server, and ensure that it returns an appropriate and timely response. This is useful in situations where it's necessary to check if a website is still functioning, particularly if there are times it comes under heavy load or suffers denial of service attacks.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `sparta.example.net`, a host defined in its own file. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

An appropriate first step is making sure that the services we intend to check are accessible from the monitoring server running Nagios Core. This can be done from the command line using an HTTP client such as `curl` or `wget`:

```
$ wget http://sparta.example.net/  
$ curl http://sparta.example.net/
```

The `check_http` plugin binary could also be called directly to test this connectivity; we'd be hoping for an HTTP OK response with a code of 200:

```
# sudo -s -u nagios  
$ /usr/local/nagios/libexec/check_http -I sparta.example.net  
HTTP OK: HTTP/1.1 200 OK - 453 bytes in 0.004 second response time  
|time=0.004264s;;;0.000000 size=453B;;;0
```

Optionally, we can check HTTPS the same way if the plugins were compiled with OpenSSL available, adding the `-S` option for the plugin:

```
# sudo -s -u nagios  
$ /usr/local/nagios/libexec/check_http -S -I sparta.example.net  
HTTP OK: HTTP/1.1 200 OK - 453 bytes in 0.058 second response time  
|time=0.057836s;;;0.000000 size=453B;;;0
```

Both may require the installation of a default page to be served by the host, probably something like `index.html`, depending on the web server software.

Once HTTP connectivity to the host from the monitoring server is verified as working with appropriate responses, we can proceed to add our service check.

# How to do it...

We can add web service checks for our host as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {
    use          linux-server
    host_name    sparta.example.net
    alias        sparta
    address      192.0.2.21
}
```

3. Beneath the definition for the host, place a new service definition for the HTTP check:

```
define service {
    use          generic-service
    host_name    sparta.example.net
    service_description HTTP
    check_command check_http
}
```

4. If an HTTPS check is also needed, add an optional second service definition:

```
define service {
    use          generic-service
    host_name    sparta.example.net
    service_description HTTPS
    check_command check_http! -S
}
```

5. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, a new service called HTTP, and optionally one called HTTPS, will be added for the `sparta.example.net` host and HTTP requests will be made from the server regularly, reporting if connectivity fails or a response comes back with an unexpected status. These services will both be visible in the Services section of the web interface.

It's important to note that, by itself, the setup given here won't check the validity of your secure certificates; it will simply check if it can make an encrypted connection.

# How it works...

The preceding configuration uses `check_http` as a plugin to make scheduled requests of the `sparta.example.net` server. By default, the root or default page is requested, so the request takes the following form:

```
GET / HTTP/1.0
User-Agent: check_http/v2.0.3 (nagios-plugins 2.0.3)
Connection: close
Accept: */*
```

The plugin awaits a response, and then returns a status based on the following criteria:

- Whether a well-formed HTTP response was received at all, within acceptable time bounds. If the response was too slow, it might raise a `CRITICAL` state when the plugin times out.
- Whether the response code for the HTTP response was `200 Found`, indicating that a document was identified and returned. A response code of `301 Moved Permanently` or `404 Not Found` would prompt a `CRITICAL` state by default.

Inspecting the command definition for the default `check_http` command in `/usr/local/nagios/etc/objects/commands.cfg` gives some insight into how it uses the plugin of the same name:

```
define command {
    command_name    check_http
    command_line    $USER1$/check_http -I $HOSTADDRESS$ $ARG1$
}
```

This uses three Nagios Core macros:

- `$USER1$`: Expands to the directory in which the plugin scripts and binaries are kept, usually `/usr/local/nagios/libexec`.
- `$HOSTADDRESS$`: The value for the address directive defined in the service's associated host, in this case `192.0.2.21`.
- `$ARG1$`: One extra argument, if defined by the command; this allows us to add the `-s` option to the `check_http` call in order to run an HTTPS check.

# There's more...

There are a great many other switches available for the `check_http` plugin; a list of them with a short explanation for each is available by entering the command with `--help`.

```
# /usr/local/nagios/libexec/check_http --help
```

One particularly useful option is `-u`, which allows us to request specific URLs other than the default index document from the server. This can be useful if we're in a situation that requires setting up checks for more than one page on a site, which can be a nice supplement to code unit testing when a site is deployed or updated.

For example, if we wanted to check that three pages were returning 200 Found responses: `about.php`, `products.php`, and `contact.php`, we could set up a new command like the following to check a specific page:

```
define command {
    command_name    check_http_page
    command_line    $USER1$/check_http -I $HOSTADDRESS$ -u $ARG1$
}
```

This would allow us to make three service checks like the following using the new command:

```
define service {
    use                generic-service
    host_name          sparta.example.net
    service_description HTTP-about
    check_command      check_http_page!/about.php
}
define service {
    use                generic-service
    host_name          sparta.example.net
    service_description HTTP-products
    check_command      check_http_page!/products.php
}
define service {
    use                generic-service
    host_name          sparta.example.net
    service_description HTTP-contact
    check_command      check_http_page!/contact.php
}
```

These service checks would run the same way as the one demonstrated in the recipe, except they would each request a specific page. Note that the leading slashes on the URLs are required.

Similarly, the `-H` option allows you to specify hostnames, which is helpful on servers hosting more than one site. This could be done by setting up a command like the following one:

```
define command {
    command_name    check_http_host
    command_line    $USER1$/check_http -I $HOSTADDRESS$ -H $ARG1$
}
```



```
}
```

This would allow you to check two sites on the same host, `http://www.example.net/` and `http://dev.example.net/`, in separate service checks:

```
define service {  
    use                generic-service  
    host_name          sparta.example.net  
    service_description HTTP-www  
    check_command       check_http_host!www.example.net  
}  
define service {  
    use                generic-service  
    host_name          sparta.example.net  
    service_description HTTP-dev  
    check_command       check_http_host!dev.example.net  
}
```

It's worth noting that the `check_http` request will show up in your server logs with its regular requests. If you're concerned about this distorting statistics, or appearing unwanted in reports, it may be easiest to filter it out using its User-Agent header value, which includes the string `check_http`.

## See also

- *Creating a new host*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new service*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new command*, [Chapter 2](#), *Working with Commands and Plugins*
- *Customizing an existing plugin*, [Chapter 2](#), *Working with Commands and Plugins*
- *The Checking a website returns a given string recipe* in this chapter



# Checking that a website returns a given string

In this recipe, we'll build on the basic web service monitoring established in the recipe *Monitoring web services* in this chapter and learn how to create a command that uses the `check_http` plugin to ensure that a particular string is included as part of an HTTP response.

By default, there's no Nagios Core command defined to use the plugin in this way, so the recipe will include defining a command before using it as part of a service check.

This may be necessary if we're monitoring a website on a server that may not necessarily return a 404 Not Found or similar error that will flag a WARNING or CRITICAL in Nagios; rather than merely checking if a document was found, we can check whether it matches a string to see if it resembles the particular document we expected.

This kind of setup is a nice complement to a suite of code unit tests for a website or web application.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `sparta.example.net`, a host defined in its own file, and we'll check that it's returning the simple string `example.net` in its responses. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

You should set up basic HTTP monitoring for the host first, as established in the *Checking web services* recipe in this chapter, to make sure that there is connectivity between the monitoring server and the host, and that requests and responses are both working correctly with appropriate error codes.

# How to do it...

We can set up a service check that includes an HTTP response content check as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit a suitable file containing command definitions and find the definition for the `check_http` command. In the default installation, this file is `commands.cfg`. The `check_http` definition looks something like this:

```
define command {
    command_name    check_http
    command_line    $USER1$/check_http -I $HOSTADDRESS$ $ARG1$
}
```

3. Beneath the `check_http` definition, add a new command definition as follows:

```
define command {
    command_name    check_http_content
    command_line    $USER1$/check_http -I $HOSTADDRESS$ -s $ARG1$
}
```

4. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {
    use              linux-server
    host_name        sparta.example.net
    alias            sparta
    address          192.0.2.21
}
```

5. Beneath the definition for the host and beneath any other checks which might use `check_http`, place a new service definition using our new command:

```
define service {
    use              generic-service
    host_name        sparta.example.net
    service_description HTTP-content
    check_command    check_http_content!example.net
}
```

6. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, Nagios Core should begin making HTTP requests to monitor the service as `check_http` normally does, except that it will only return an OK state if the content of the website includes the string `example.net`. Otherwise, it will generate an alert and flag the

service as CRITICAL, with a message like the following:

```
HTTP CRITICAL: HTTP/1.1 200 OK - string 'example.net' not found on  
'http://192.0.2.21:80/' - 453 bytes in 0.006 second response time
```

## How it works...

One of the many options for `check_http` is `-s`, short for `--string`, which takes a single argument specifying a string that must occur in the content for the service check to return an OK state.

When the HTTP response is received, `check_http` examines the text in the response to see if it matches the string specified, on top of its usual behavior of flagging `WARNING` or `CRITICAL` states for connectivity or timeout problems.

Note that, in order to make this work, it was necessary to define a new command that uses the first argument (in this case, the string `example.net`) as the value for the `-s` option to `check_http`. The full command line executed would look something like the following one:

```
# /usr/local/nagios/libexec/check_http -H sparta.example.net -s example.net
```



## There's more...

The `check_http` plugin allows considerably more than single string checks; if it's necessary to test for the presence of a regular expression in the content, this can also be done using the `-r` or `--regex` options. We could define a command to check for regular expressions like this:

```
define command {  
    command_name    check_http_regex  
    command_line    $USER1$/check_http -I $HOSTADDRESS$ -r $ARG1$  
}
```

If it's necessary to check that a particular regular expression doesn't match the content, this is possible by adding the `--invert-regex` flag:

```
define command {  
    command_name    check_http_noregex  
    command_line    $USER1$/check_http -I $HOSTADDRESS$ -r $ARG1$ --invert-  
regex  
}
```

A service check using this command would return `CRITICAL` if the response was found to match the pattern provided as the first argument to a `check_command` directive.

Other similar options include `-e` or `--expect`, which allows specifying a comma-separated set of strings, at least one of which must match the content for the check to pass.

## See also

- *Creating a new host*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new service*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new command*, [Chapter 2](#), *Working with Commands and Plugins*
- *Customizing an existing plugin*, [Chapter 2](#), *Working with Commands and Plugins*
- The *Monitoring web services* recipe in this chapter



# Monitoring database services

In this recipe, we'll learn how Nagios Core can be used to monitor the status of a database server. We'll demonstrate with the popular MySQL as an example, using the `check_mysql` plugin, and we'll discuss running an actual test query and specifying a similar check for PostgreSQL in the *There's more...* section.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. We'll use the example of `delphi.example.net`, a host defined in its own file. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

For a check on a remote host to work from the monitoring server, the database server will need to be listening on an appropriate network interface. It's also necessary to make sure that an appropriate database user account exists with which the `check_mysql` plugin may authenticate. It's a good idea to make this into a dedicated user with no privileges on any database, because the credentials need to be stored in plain text, which could be a security risk if more sensitive credentials were used.

For MySQL, we could create a new user with no privileges with the following, assuming that the monitoring server `olympus.example.net` has `192.0.2.11` as an IPv4 address. I've used a randomly generated password here:

```
mysql> CREATE USER 'nagios'@'192.0.2.11' IDENTIFIED BY 'UVocjPHoH0';
```

We can then check the connectivity using the `mysql` client on the monitoring server:

```
$ mysql --host=delphi.example.net --user=nagios --password=UVocjPHoH0
mysql>
```

Or alternatively, by running the plugin directly from the command line as the `nagios` user:

```
# sudo -s -u nagios
$ /usr/local/nagios/libexec/check_mysql -H delphi.example.net -u nagios -p
UVocjPHoH0
Uptime: 1631  Threads: 1  Questions: 102  Slow queries: 0  Opens: 99
Flush tables: 1  Open tables: 23  Queries per second avg: 0.62
```

Newer versions of the plugins may also include extra performance information after the preceding output.

If we did not have the MySQL libraries installed when we built the Nagios plugins, we may find that we do not have the `check_mysql` and `check_mysql_query` binaries in `/usr/local/nagios/libexec`. We can fix this by installing the MySQL shared libraries on the monitoring system and rebuilding and reinstalling the Nagios Plugins package.

By default, it's also necessary to define new commands to actually use these plugins as well, which we'll do in this recipe.

# How to do it...

We can set up some basic database monitoring for our MySQL server as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit a suitable file containing command definitions, perhaps `commands.cfg`, and add the following definition:

```
define command {  
    command_name    check_mysql  
    command_line    $USER1$/check_mysql -H $HOSTADDRESS$ -u $ARG1$ -p  
$ARG2$  
}
```

3. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {  
    use                linux-server  
    host_name          delphi.example.net  
    alias              delphi  
    address            192.0.2.51  
}
```

4. Beneath the definition for the host, place a new service definition for the MySQL check, including the username and password chosen earlier for arguments:

```
define service {  
    use                generic-service  
    host_name          delphi.example.net  
    service_description MySQL  
    check_command       check_mysql!nagios!UVocjPHoH0  
}
```

5. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, a new service check with the description `MySQL` will be added for the `delphi.example.net` host, which will employ the `check_mysql` plugin to report the status of the MySQL server. The output will also include statistics about its uptime, open tables, and query averages, and will be visible in the web interface under **Services** like all service output.

## How it works...

This configuration defines a new command named `check_mysql` to use the plugin of the same name, accepting two arguments—the first being the username of the test Nagios Core user, in this case `nagios`, and the second being the password for that user. The `check_mysql` plugin acts as a MySQL client using the credentials provided to it and requests diagnostic information from the database, which it returns as part of its check.

If it has problems connecting to or using the MySQL server, it will flag a status of `CRITICAL` and generate appropriate notifications.

## There's more...

We can optionally check access to a specific database using the plugin by supplying a value to the `-d` parameter. This should be a database to which the nagios user has been given access, otherwise the check would fail.

If we want to check that we can actually run a query after connecting, we could extend this even further to use the `check_mysql_query` plugin:

```
define command {
    command_name    check_mysql_query
    command_line    $USER1$/check_mysql_query -H $HOSTADDRESS$ -u $ARG1$ -p
$ARG2$ -d $ARG3$ -q $ARG4$
}
define service {
    use                generic-service
    host_name          delphi.example.net
    service_description MYSQL_QUERY
    check_command
check_mysql_query!nagios!UVocjPHoH0!exampledb!"SELECT COUNT(1) FROM
exampletbl"
}
```

The preceding code would attempt to run the query `SELECT COUNT(1) FROM exampletbl` on the `exampledb` database. Note that it is important to wrap the query in quotes so that it gets processed as one argument rather than several.

A similar service check to the one specified in this recipe could be configured for PostgreSQL database servers using the `check_pgsq1` plugin, also part of the standard Nagios Plugins set. The command and service check definitions might look something like this:

```
define command {
    command_name    check_pgsq1
    command_line    $USER1$/check_pgsq1 -H $HOSTADDRESS$ -p $ARG1$
}
define service {
    use                generic-service
    host_name          delphi.example.net
    service_description PGSQ1
    check_command     check_pgsq1!N4Nw8o8X
}
```

In the preceding example, access would need to be granted on the PostgreSQL server for the monitoring server's IP address in `pg_hba.conf`, with access to the default standard `template1` database.

In production environments, it's often the case that, for security or programming policy reasons, database servers are not actually configured to accept direct connections over network interfaces, even on secure interfaces. Packaged MySQL and PostgreSQL servers on many systems will, in fact, default to listening only on the `localhost` interface on `127.0.0.1`.



This can complicate the monitoring setup a little, but it can usually be addressed by installing a remote Nagios plugin execution agent on the database server, such as NRPE or NSclient++. NRPE usage is addressed in [Chapter 6](#), *Enabling Remote Execution*, of this book and uses a MySQL server configured in this way as a demonstration of the concept.

## See also

- *Creating a new host, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new command, [Chapter 2](#), Working with Commands and Plugins*
- *Monitoring local services on a remote machine with NRPE, [Chapter 6](#), Enabling Remote Execution*



# Monitoring the output of an SNMP query

In this recipe, we'll learn how to use the `check_snmp` plugin to monitor the output made by SNMP (Simple Network Management Protocol) requests.

Despite its name, SNMP is not really a very simple protocol. However, it's a very common method for accessing information on many kinds of networked devices, including monitoring boards, usage meters, and storage appliances as well as workstations, servers, and routing equipment.

Because SNMP is so widely supported and typically able to produce such a large volume of information to trusted hosts, it's an excellent way of gathering information from hosts that's not otherwise retrievable from network services. For example, while checking for a PING response from a large router is simple enough, there may not be an easy way to check properties like the state of each of its interfaces, or the presence of a certain route in its routing tables.

Using `check_snmp` in Nagios Core allows automated retrieval of this information from the devices and generating alerts appropriately. While its setup is somewhat complex, it is worth learning how to use it as it is among the most powerful plugins in Nagios Core for network administrators and it is quite typical to see dozens of commands defined for its use in a typical configuration for a large network. It can often be used to complement or even replace remote plugin execution daemons like NRPE or NSClient++.

# Getting ready

You should have a Nagios Core 4.0 or newer server with at least one host configured already. You should also understand the basics of how hosts and services relate, which is covered in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

This recipe assumes a basic knowledge of SNMP, including its general intended purpose, the concept of an SNMP community, and what SNMP MIBs and OIDs are. In particular, if you're looking to monitor some property of a networked device that's available to you via SNMP, you should know what the OID for that data is. This information is often available in the documentation for network devices, or can be deduced by running an appropriate `snmpwalk` command against the host to view the output for all its OIDs.

You should check that an SNMP daemon is running on the target host and also that the `check_snmp` plugin is available on the monitoring host. It is included as part of the standard Nagios Plugins so, provided the Net-SNMP libraries were available on the system when these were compiled, it should be available. If it is not, you may need to install the Net-SNMP libraries on your monitoring system and recompile the plugins.

We'll use the example of retrieving the total process count from a Linux server with hostname `ithaca.example.net` and flagging `WARNING` and `CRITICAL` states at appropriate high ranges. We'll also discuss how to test for the presence or absence of strings rather than numeric thresholds.

It's a good idea to test that the host will respond to SNMP queries in the expected form. We can test this with `snmpget`. Assuming a community name of `public`, we could write:

```
$ snmpget -v1 -c public ithaca.example.net .1.3.6.1.2.1.25.1.6.0
iso.3.6.1.2.1.25.1.6.0 = Gauge32: 81
```

We can also test the plugin by running it directly as the `nagios` user:

```
# sudo -s -u nagios
$ /usr/local/nagios/libexec/check_snmp -H ithaca.example.net -C public -o
.1.3.6.1.2.1.25.1.6.0
SNMP OK - 81 | iso.3.6.1.2.1.25.1.6.0=81
```

# How to do it...

We can define a command and service check for the Linux process count OID as follows:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to that directory instead.

```
# cd /usr/local/nagios/etc/objects
```

2. Edit a suitable file containing command definitions, perhaps `commands.cfg`, and add the following definition to the end of the file.

```
define command {  
    command_name    check_snmp_linux_procs  
    command_line    $USER1$/check_snmp -H $HOSTADDRESS$ -C $ARG1$ -o  
                    .1.3.6.1.2.1.25.1.6.0 -w 100 -c 200  
}
```

3. Edit the file containing the definition for the host. The host definition might look something like this:

```
define host {  
    use                linux-server  
    host_name          ithaca.example.net  
    alias              ithaca  
    address            192.0.2.61  
}
```

4. Beneath the definition for the host, place a new service definition using our new command. Replace `public` with the name of your SNMP community if it differs:

```
define service {  
    use                generic-service  
    host_name          ithaca.example.net  
    service_description SNMP_PROCS  
    check_command      check_snmp_linux_procs!public  
}
```

5. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, a new service check with a description of `SNMP_PROCS` will be added to the `ithaca.example.net` host and the `check_snmp` plugin will issue a request for the value of the specified OID as its regular check. It will flag a `WARNING` state if the count is greater than 100 and a `CRITICAL` state if greater than 200, notifying accordingly. All this appears in the web interface the same way as any other service, under the **Services** menu item.

## How it works...

The preceding configuration defines both a new command based around the `check_snmp` plugin and, in turn, a new service check using that command for the `ithaca.example.net` server. The community name for the SNMP request, `public`, is passed into the command as an argument; everything else, including the OID to be requested, is fixed into the `check_snmp_linux_procs` command definition.

Part of the command line defined includes the `-w` and `-c` options. For numeric outputs like ours, these are used to define the limits for the value beyond which a `WARNING` or `CRITICAL` state is raised, respectively. In this case, we define a `WARNING` threshold of 100 processes and a `CRITICAL` threshold of 200 processes.

Similarly, if the SNMP check fails completely due to connectivity problems or syntax errors, an `UNKNOWN` state will be reported.

## There's more...

It's also possible to test the output of SNMP checks to see if they match a particular string or pattern for determining whether the check succeeded. If we needed to check that the system's short hostname was `ithaca`, for example (perhaps as a simple test SNMP query that should always succeed), we might set up a command definition as follows:

```
define command {
    command_name    check_snmp_hostname
    command_line    $USER1$/check_snmp -H $HOSTADDRESS$ -C $ARG1$ -o
.1.3.6.1.2.1.1.5.0 -r $ARG2$
}
```

With a corresponding service check like this:

```
define service {
    use                generic-service
    host_name          ithaca.example.net
    service_description SNMP_HOSTNAME
    check_command       check_snmp_hostname!public!ithaca
}
```

This particular check would only succeed if the SNMP query succeeds and returns a string matching the string `ithaca`, as specified in the second argument.



## See also

- *Creating a new host*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new service*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new command*, [Chapter 2](#), *Working with Commands and Plugins*
- The *Creating an SNMP OID to monitor* recipe in this chapter



# Monitoring a RAID or other hardware device

In this recipe, we'll learn a general strategy for monitoring the properties of hardware devices. Because of the different ways that vendors implement their hardware, this tends to be less straightforward than monitoring standard network services.

There are at least four general approaches to this problem.

# Getting ready

You will need to know some specifics about the hardware that you want to monitor, including the model number. You should preferably also have a Nagios Core 4.0 server that was compiled with Net-SNMP libraries available to build the `check_snmp` plugin.

# How to do it...

We can find a way to monitor an arbitrary hardware device on a local or remote machine as follows:

1. Check if the official or unofficial Nagios Core plugins already exist for polling the particular device. The best place to start is with Nagios Exchange at <https://exchange.nagios.org/>; just search for the make of the hardware and see if a plugin already exists, as per the *Finding a plugin* recipe in [Chapter 2, Working with Commands and Plugins](#). You can then install it with *Installing a plugin* recipe in the same chapter.
2. Check if any of the values you need from the hardware are or can be exported as SNMP OIDs, to be checked with the recipe *Monitoring the output of an SNMP query* in this chapter.
3. If they aren't, but there's a command line diagnostic tool, the output or return value of which can be used as the check, you could consider exporting it as a custom OID in an SNMP server with the *Creating a new SNMP OID to monitor* recipe in this chapter.
4. Finally, we may have to resort to writing our own plugin. This is not actually as difficult as it may seem; it is discussed in the recipe *Writing a new plugin from scratch* in [Chapter 2, Working with Commands and Plugins](#). This may be the only option for custom or very uncommon hardware.

## How it works...

If we find an appropriate plugin for the hardware online, the main snag here is that we will need to be sure not only that the plugin works by testing it against the hardware in both an OK and a CRITICAL state (which might be hard to do), but we also need to make sure the plugin is safe to run. The plugins on Nagios Exchange are reviewed before they are added, but the code for plugins that you find on any other website might not be safe to run.

Using SNMP for these kinds of checks where possible has two advantages in that the values can be checked using a standard Nagios plugin, `check_snmp`, and the values can also be read over the network, meaning that we may not need to rely on remote execution daemons like NRPE or NSclient++ to get this information.

## See also

- The *Monitoring the output of an SNMP query* recipe in this chapter
- The *Creating a new SNMP OID to monitor* recipe in this chapter
- *Finding a plugin*, [Chapter 2](#), *Working with Commands and Plugins*
- *Installing a plugin*, [Chapter 2](#), *Working with Commands and Plugins*
- *Writing a new plugin from scratch*, [Chapter 2](#), *Working with Commands and Plugins*





# Creating an SNMP OID for monitoring

In this recipe, we'll learn how to configure a Net-SNMP `snmpd` server on a Linux server to return the output of a command in an SNMP OID. This can be useful as an alternative to NRPE monitoring for information that is not otherwise available in a checkable network service so that Nagios Core can check it via its standard `check_snmp` method.

As an example, this can be a very good way of monitoring hardware devices such as RAID arrays on remote servers, where command line diagnostic tools are available for reporting a status as a number or string, but those only work locally, and don't otherwise include any information in an SNMP MIB tree.

# Getting ready

The host we intend to check should be running a Net-SNMP `snmpd` server that allows full read access to the MIB tree for a specified community string such as `public`. This SNMP server should be capable of using the `exec` directive in its configuration to return the output of a command as the value of an SNMP OID when requested by an SNMP client. As such, you will need to know the basics of SNMP.

You will also need a Nagios Core 4.0 or later server with SNMP enabled in order to actually monitor the OID, which is explained in the recipe *Monitoring the output of an SNMP query*.

In this example, we'll deal with a simple script running on a target host `ithaca.example.net` called `/usr/local/bin/raidstat` that returns an integer: zero for the RAID being in a good state, and non-zero to signal any problems. This tool can be run as any user.

# How to do it...

We can set up our custom SNMP OID as follows:

1. Add the following line to the `snmpd.conf` file on the target host, substituting in the name of the command that generates the needed output:

```
exec raidstat /usr/local/bin/raidstat
```

2. Restart the `snmpd` server on the target host, which might be done as follows, depending on the system:

```
# /etc/init.d/snmpd restart
```

3. Back on the monitoring server, we can use the `.1.3.6.1.4.1.2021` OID, and find our `raidstat` OID and its value in the output:

```
$ snmpwalk -v1 -c public ithaca.example.net .1.3.6.1.4.1.2021 | less  
...  
iso.3.6.1.4.1.2021.8.1.2.1 = STRING: "raidstat"  
iso.3.6.1.4.1.2021.8.1.3.1 = STRING: "/usr/local/bin/raidstat"  
iso.3.6.1.4.1.2021.8.1.100.1 = INTEGER: 0  
iso.3.6.1.4.1.2021.8.1.101.1 = STRING: "GOOD"  
...
```

4. We now know which OID we can query for a check using the *Monitoring the output of an SNMP query* recipe in this chapter and can test retrieving it directly with `snmpget`:

```
$ snmpget -v1 -c public ithaca.example.net 1.3.6.1.4.1.2021.8.1.100.1
```

## How it works...

The exec call in the snmpd configuration defines a program that should be run to return a new value in the OID tree. The four OIDs we found in the output, which return data about our process, are chosen for us by snmpd:

- 1.3.6.1.4.1.2021.8.1.2.1: The name of the OID we've assigned, raidstat
- 1.3.6.1.4.1.2021.8.1.3.1: The full path to the script that was called, /usr/local/bin/raidstat
- 1.3.6.1.4.1.2021.8.1.100.1: The return value from the call, expressed as an integer
- 1.3.6.1.4.1.2021.8.1.101.1: Any output from the call, expressed as a string

This setup can therefore be used to check the return value and character output of any command that the snmpd user is able to execute, a good method for ad hoc monitoring for very specific cases like this.

## There's more...

More than one `exec` command can be added to the server configuration if required. For example, if we needed to check the state of the CPU temperature with a command `tempstat`, we might define it as follows:

```
exec raidstat /usr/local/bin/raidstat
exec tempstat /usr/local/bin/tempstat
```

The return values and output for both would then show up in the SNMP output in separate OIDs.

If necessary, the command definitions can also be followed by arguments:

```
exec tempstat /usr/local/bin/tempstat --cpu=0
```

A full discussion of how `exec` and the similar `pass` configuration for Net-SNMP works is outside the scope of this book, but is discussed extensively in its documentation, available at the time of writing at <http://www.net-snmp.org/docs/readmefiles.html>.

Note that if exporting values via SNMP is unsuitable, an alternative for remote monitoring is to use NRPE (Nagios Remote Plugin Executor) to run a check on the target server and return the result to the monitoring server. This is discussed in the recipes in [Chapter 6](#), *Enabling Remote Execution*.

## See also

- *Creating a new host, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new command, [Chapter 2](#), Working with Commands and Plugins*
- *The Monitoring the output of an SNMP query recipe in this chapter*
- *Monitoring local services on a remote machine with NRPE, [Chapter 6](#), Enabling Remote Execution*



# Chapter 6. Enabling Remote Execution

In this chapter, we will cover the following recipes:

- Monitoring local services on a remote machine with NRPE
- Setting the listening address for NRPE
- Setting allowed client hosts for NRPE
- Creating new NRPE command definitions securely
- Giving limited `sudo(8)` privileges to NRPE
- Using `check_by_ssh` with key authentication instead of NRPE
- Using `check_mk` instead of NRPE



# Introduction

For a dedicated Nagios Core server with access to all the relevant parts of the network, making checks is relatively simple using commands and plugins that make ICMP, TCP, and UDP connections to network hosts and services for determining their operating state. These can be used to check any sort of network service, without requiring anything to be installed on the target machine. As an example, when the `check_http` plugin is used to check a web server, it works the same way as a browser making the request.

However, monitoring a network thoroughly usually has more to it than simply checking network connectivity and availability. It's also a good idea to check properties of the network that don't directly correspond to a network service and hence can't be directly checked over a network connection.

These are often properties of hardware or the underlying system, such as disk space or system load average, or processes that are configured only to listen locally, commonly done for database servers.

We could install Nagios Core on all the systems, perhaps, but this would make maintenance difficult. It would be much better to have some means of remotely executing diagnostic programs so that they are run directly on the target host to retrieve the information they need and the results are returned to a single Nagios Core server via a dedicated network service.

There are four general approaches to managing this problem:

- Use `check_nrpe` to run a standard Nagios Core plugin on the target machine and return its results transparently to the monitoring server
- Use `check_by_ssh` to run an arbitrary command on the target machine from the monitoring server by first connecting to it with SSH
- Use `check_mk` to automatically specify and run a set of commands appropriate to the target machine and return the results transparently to the monitoring server
- Use `check_snmp` to check an SNMP OID that's configured to provide the return value and output of some command on the target host

This chapter covers the first three solutions, focusing on the more commonly used **Nagios Remote Plugin Executor (NRPE)**, and explaining how it differs from the `check_by_ssh` solution. For some information on configuring SNMP, see the recipes *Monitoring the output of an SNMP query* and *Creating an SNMP OID to monitor*, both in [Chapter 5, Monitoring Methods](#).



# Monitoring local services on a remote machine with NRPE

In this recipe, we'll learn how to install and run an NRPE (Nagios Remote Plugin Executor) server on a target host, `roma.example.net`. We'll use this to check the load average on that host with the `check_load` plugin.

The plugins for these checks will be executed on the target server by the NRPE daemon, but the results will be returned to our Nagios Core monitoring server, `olympus.example.net`. This requires installing the `check_nrpe` plugin on the monitoring server and the full Nagios Plugins set (but not Nagios Core itself) on the target server.

This is a reasonably long and in-depth recipe as it involves installing a total of three software packages on two servers.

# Getting ready

You will need a monitoring server with Nagios Core 4.0 or newer installed. You should also have a UNIX-like target host that you intend to monitor that can run the NRPE daemon. Most modern UNIX-like systems including Linux and BSD should be able to do this. Both the monitoring server and the target host will need internet connectivity and you should already be monitoring the target host itself with a host definition to which we'll be adding service checks.

If your servers don't have a direct gateway to the internet, you can work around this by uploading the relevant files after downloading them onto a workstation, or another machine with internet access.

You should understand the basics of configuring, compiling, and installing software from source. In most cases, a simple `./configure`, `make`, and `make install` will be all that's necessary and the recipe will walk you through this. You will need to have `make(1)` installed along with any other tools needed for the process, including a C compiler like `gcc(1)`.

You should also have a good grasp on how hosts and services interrelate in Nagios Core, which is discussed in the recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*, and how Nagios Core uses commands and plugins, discussed in [Chapter 2](#), *Working with Commands and Plugins*. You should not need an in-depth understanding of the use of any particular plugin; the recipe will demonstrate the usage of the plugins to which it refers.

Finally, you should be able to configure any firewalls to allow connectivity from the Nagios Core server to the server being monitored with the TCP destination port 5666.

# How to do it...

This first part of the recipe is done on the target server.

1. Download and inflate the latest Nagios Plugins package. At the time of writing, the link is available at <http://nagiosplugins.org/download/>.

```
$ wget http://nagios-plugins.org/download/nagios-plugins-2.1.1.tar.gz
$ tar -xzf nagios-plugins-2.1.1.tar.gz
```

2. Configure, compile, and install the plugins, the same way you would on a new monitoring server. You will need to have root privileges for the command `make install` call.

```
$ cd nagios-plugins-2.1.1
$ ./configure
$ make
# make install
```

You may need to install some shared libraries and headers on the system to do this for certain plugins, such as a `libssl` implementation. The output of the `./configure` script should alert you of any such problems.

3. Download and inflate the latest version of NRPE from the Nagios Exchange website. At the time of writing, the link is available at:  
<http://exchange.nagios.org/directory/Addons/Monitoring-Agents/NRPE—2D-Nagios-Remote-Plugin-Executor/details>.

```
$ wget http://downloads.sourceforge.net/project/nagios/nrpe-2.x/nrpe-2.15/nrpe-2.15.tar.gz
$ tar -xzf nrpe-2.15.tar.gz
```

4. Enter the `nrpe-2.15` source directory and configure, compile, and install the daemon and a stock configuration for it. You will need to have root privileges for both the `make install-daemon` and the `make install-daemon-config` calls.

```
$ cd nrpe-2.15
$ ./configure
$ make all
# make install-daemon
# make install-daemon-config
```

If you do not already have a `nagios` user on the target host, you may need to create one before the daemon installs properly:

```
# groupadd nagios
# useradd -r -g nagios nagios
```

5. Edit the newly installed file at `/usr/local/nagios/etc/nrpe.cfg` and find the line beginning with `allowed_hosts`. Add a comma and the IP address of your monitoring server to this line. In this case, we've added the IP address `192.0.2.11`:

```
allowed_hosts=127.0.0.1,192.0.2.11
```

6. Start the nrpe daemon and check it's running by searching the process table with `pgrep(1)` or `ps(1)`:

```
# /usr/local/nagios/bin/nrpe -c /usr/local/nagios/etc/nrpe.cfg -d
# pgrep nrpe
18593
# ps -e | grep [n]rpe
nagios 18593 1 0 21:55 ? 00:00:01 nrpe
```

If you would like the nrpe daemon to start on boot, add an init script appropriate to your system. An example init-script is generated at `./configure` time in the source directory. Versions are also generated for Debian-derived systems and SUSE systems in `init-script.debian` and `init-script.suse` respectively. Exactly how this should be done will depend on your particular system, for which you may need to consult its documentation.

This next part of the recipe is done on the monitoring server.

1. Again, download the latest version of NRPE, the same way as was done for the target server:

```
$ wget http://downloads.sourceforge.net/project/nagios/nrpe-2.x/nrpe-2.15/nrpe-2.15.tar.gz
$ tar -xzf nrpe-2.15.tar.gz
```

2. Again, configure and build the software. However, note that this time the `install` line is different, as we're installing the `check_nrpe` plugin rather than the nrpe daemon:

```
$ cd nrpe-2.15.tar.gz
$ ./configure
$ make all
# make install-plugin
```

3. Check that the plugin has been correctly installed. It should be saved at `/usr/local/nagios/libexec/check_nrpe`:

```
$ ls /usr/local/nagios/libexec/check_nrpe
/usr/local/nagios/libexec/check_nrpe
```

4. Move to the directory containing the Nagios Core object configuration. By default, this is `/usr/local/nagios/etc/objects`.

```
$ cd /usr/local/nagios/etc/objects
```

5. Edit an appropriate file for defining new commands. For the default installation, `/usr/local/nagios/etc/objects/commands.cfg` is a good choice. Add the following definition to the end of this file:

```
define command {
    command_name    check_nrpe
    command_line    $USER1$/check_nrpe -H $HOSTADDRESS$ -c $ARG1$
}
```

6. Edit the file defining the target host as an object. The definition might look something like this:

```
define host {  
    use          linux-server  
    host_name    roma.example.net  
    alias        roma  
    address      192.0.2.61  
}
```

7. Beneath the definition for the host, after any other services defined for it, add the following service definition:

```
define service {  
    use          generic-service  
    host_name    roma.example.net  
    service_description    LOAD  
    check_command    check_nrpe!check_load  
}
```

8. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios reload
```

With this done, a new service with the description `LOAD` should appear in the web interface, ready to be checked, and with an appropriate status, including the load average as read from the `nrpe` daemon on the target host, as shown in the following screenshot:

|                  |      |    |                     |              |     |                                     |
|------------------|------|----|---------------------|--------------|-----|-------------------------------------|
| roma.example.net | LOAD | OK | 08-05-2015 21:26:44 | 0d 0h 1m 39s | 1/3 | OK - load average: 0.00, 0.17, 0.30 |
|------------------|------|----|---------------------|--------------|-----|-------------------------------------|

We can see more detail about how the check was performed and its results in the details page for the service:

### Service State Information

**Current Status:** **OK** (for 0d 0h 3m 50s)  
**Status Information:** OK - load average: 0.00, 0.17, 0.30  
**Performance Data:** load1=0.000;15.000;30.000;0; load5=0.170;  
10.000;25.000;0; load15=0.300;5.000;20.000;0;  
**Current Attempt:** 1/3 (HARD state)  
**Last Check Time:** 08-05-2015 21:26:44  
**Check Type:** ACTIVE  
**Check Latency / Duration:** 0.000 / 0.047 seconds  
**Next Scheduled Check:** 08-05-2015 21:36:44  
**Last State Change:** 08-05-2015 21:26:44  
**Last Notification:** N/A (notification 0)  
**Is This Service Flapping?** **NO** (0.00% state change)  
**In Scheduled Downtime?** **NO**  
**Last Update:** 08-05-2015 21:30:31 ( 0d 0h 0m 3s ago)

**Active Checks:** **ENABLED**  
**Passive Checks:** **ENABLED**  
**Obsessing:** **ENABLED**  
**Notifications:** **ENABLED**  
**Event Handler:** **ENABLED**  
**Flap Detection:** **ENABLED**

If the load average on roma.example.net exceeds the limits defined for the check\_load command in /usr/local/nagios/etc/nrpe.cfg on the target host, the service will enter WARNING or CRITICAL states, and will send notifications if configured to do so, all in the same manner as a non-NRPE service.



# How it works...

The NRPE plugin and daemon are used to run Nagios Core plugins on the target host rather than on the monitoring server itself. The results of the check are then passed back to the monitoring server and recorded and analyzed by Nagios Core the same way as if the service was running a plugin on the monitoring server, for example, `check_http` or `check_ssh`.

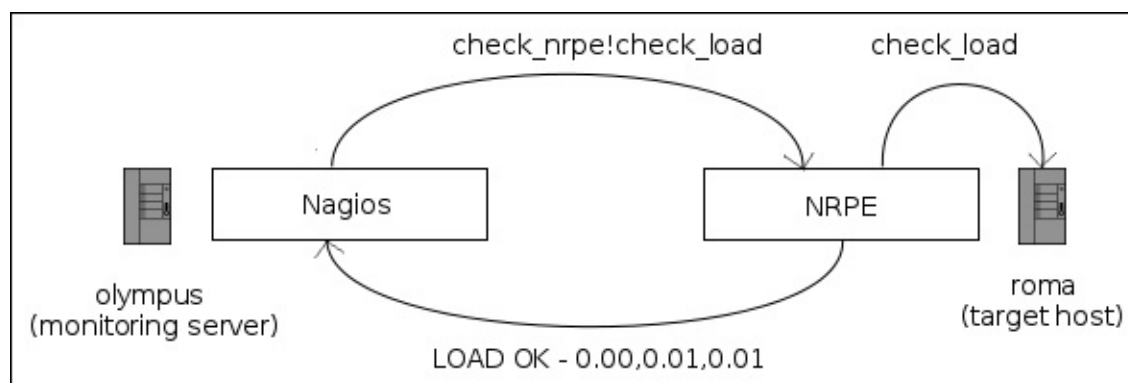
The recipe we followed does four main things:

- We installed the latest Nagios Plugins package to the target host, including the `check_load` plugin. This is necessary, because the plugin is actually run on the target host and not on the monitoring server as is the case with plugins that check network services.
- We installed the `nrpe` daemon to the target host along with a stock configuration file `nrpe.cfg`. This is the network service through which the `check_nrpe` plugin will request commands to be run on the target host. The plugins will be run by this process, typically as the `nagios` user.
- We installed the `check_nrpe` plugin to the monitoring host and defined a command of the same name to use it. The command accepts one argument in the `$ARG1$` macro; its value is the command that should be run on the target host. In this case, we supplied `check_load` for this argument.
- We set up a service to monitor the output of the standard `check_load` plugin via `check_nrpe`.

Like other Nagios Core plugins, the `check_nrpe` program can be run directly from the command line. If we wanted to test the response of the configuration we've arranged in the preceding recipe, we might run the following:

```
$ /usr/local/nagios/libexec/check_nrpe -H roma.example.net -c check_load
OK - load average: 0.00, 0.00, 0.00|load1=0.000;15.000;30.000;0;
load5=0.000;10.000;25.000;0; load15=0.000;5.000;20.000;0;
```

In this case, the state of `OK` and the load average values, as retrieved by `check_load`, were returned by the `nrpe` daemon as the result of the `check_nrpe` call.



It's very important to note that this simple configuration of NRPE is not completely secure

by default. The recipes listed under *See also* for this recipe provide some basic means to secure NRPE instances from abuse. These should be used in concert with a sensible firewall policy.

## There's more...

Of course, `check_load` is not the only plugin that can be run on the target server this way. If we inspect the file `/usr/local/nagios/etc/nrpe.cfg` on the target host, near the end of the file we find some other example definitions of commands that `check_nrpe` will run upon requests issued from the monitoring server:

```
command[check_users]=/usr/local/nagios/libexec/check_users -w 5 -c 10
command[check_load]=/usr/local/nagios/libexec/check_load -w 15,10,5 -c
30,25,20
command[check_hda1]=/usr/local/nagios/libexec/check_disk -w 20% -c 10% -p
/dev/hda1
command[check_zombie_procs]=/usr/local/nagios/libexec/check_procs -w 5 -c
10 -s Z
command[check_total_procs]=/usr/local/nagios/libexec/check_procs -w 150 -c
200
```

We recognize `check_load` as the second of these. Note that it already includes some thresholds for `WARNING` and `CRITICAL` alerts in its `-w` and `-c` parameters.

If we also wanted to check the number of processes on this server, we could add a service check for `roma.example.net`, defined like this:

```
define service {
    use                generic-service
    host_name          roma.example.net
    service_description PROCS
    check_command       check_nrpe!check_total_procs
}
```

This service will generate a `WARNING` alert if the number of processes exceeds 150 and a `CRITICAL` alert if it exceeds 200. Again, the plugin is run on the target server and not the monitoring server.

Another useful and common application of `check_nrpe` is to make remote checks on database servers with plugins like `check_mysql` and `check_pgsql` in the case where the servers do not listen on network interfaces for security reasons. Instead, they listen only on `localhost` or `UNIX` sockets and are hence inaccessible to the monitoring server. To work around this problem, we could add a new command definition to the end of `nrpe.cfg` on the target server as follows:

```
command[check_mysql]=/usr/local/nagios/libexec/check_mysql -u nagios -d
nagios -p wGG7H233bq
```

A corresponding check that uses the `check_mysql` command can then be made on the monitoring server:

```
define service {
    use                generic-service
    host_name          roma.example.net
    service_description MYSQL
    check_command       check_nrpe!check_mysql
}
```

See the *Monitoring database services* recipe in [Chapter 5, Monitoring Methods](#), for some detail on how to use the `check_mysql` and `check_pgsql` plugins.

NRPE is thus useful not just for making checks of system properties or hardware, but for any plugin that needs to be run on the target host rather than on the monitoring host.

Finally, it's important to note that the command definitions included in the default `nrpe.cfg` file are intended as examples; you will probably want to fine-tune the parameters for some of them and remove the ones you don't use along with adding your own.

## See also

- The *Setting the listening address for NRPE* recipe in this chapter
- The *Setting allowed client hosts for NRPE* recipe in this chapter
- The *Creating new NRPE command definitions securely* recipe in this chapter
- The *Giving limited sudo(8) privileges to NRPE* recipe in this chapter
- *Monitoring database services*, [Chapter 5](#), *Monitoring Methods*



# Setting the listening address for NRPE

In this recipe, we'll learn how to make NRPE listen on a specific IP address on a target host. This might be done on hosts with multiple interfaces in order to prevent spurious requests made to the `nrpe` daemon from untrusted interfaces, perhaps the public Internet. It could also be appropriate for making the daemon only listen on a trusted VPN interface.

This setup can be particularly useful when the server has an interface into a dedicated management network to which the monitoring server also has access, preventing the `nrpe` daemon from responding to requests on other interfaces unnecessarily and thereby closing a possible security hole.

# Getting ready

You should have a target host configured for checking in a Nagios Core 4.0 or later monitoring server. The target host should be running the nrpe daemon and listening on all interfaces (which we'll fix). You can verify that nrpe is running with `pgrep(1)` or `ps(1)`:

```
# pgrep nrpe
29964
# ps -e | grep [n]rpe
nagios 29964 1 0 21:55 ? 00:00:01 nrpe
```

You can check whether the nrpe daemon is listening on all interfaces by checking the output of `netstat(8)` or `ss(8)`:

```
# netstat -plnt | grep nrpe
tcp 0 0 0.0.0.0:5666 0.0.0.0:* LISTEN 29964/nrpe
# ss -l | grep nrpe
tcp LISTEN 0 5 *:nrpe *:*
tcp LISTEN 0 5:::nrpe :::*
```

The address of `0.0.0.0` or `::` shows that nrpe is listening on all interfaces, which is what we'd like to correct.



# How to do it...

We can configure the nrpe daemon to only listen on one address as follows:

1. Edit the nrpe daemon's configuration file. The default location is `/usr/local/nagios/etc/nrpe.cfg`. Look for the line beginning with `server_address`, which is normally commented out by default:

```
#server_address=127.0.0.1
```

If you don't have such a line, you can add it at the end of the file.

2. Uncomment the line if it's commented by removing the leading `#` character and change the `127.0.0.1` address to the address to which you want to restrict the nrpe process listening:

```
server_address=192.0.2.61
```

3. Restart the nrpe daemon. If you have installed an `init` script for it, you may be able to do this with something like:

```
# /etc/init.d/nrpe restart
```

If not, you can restart the process by sending it an HUP signal with the `pkill(1)` command, which will prompt it to re-read its configuration file and resume running:

```
# pkill -HUP nrpe
```

With this done, the nrpe daemon should now only be listening on the specified address. We can verify this using `netstat(8)` or `ss(8)`:

```
# netstat -plnt | grep nrpe
tcp 0 0 192.0.2.61:5666 0.0.0.0:* LISTEN 29964/nrpe
# ss -l | grep nrpe
tcp LISTEN 0 5 192.0.2.61:nrpe *:*
```

## How it works...

The configuration we adjusted defines an address on which the `nrpe` daemon should listen and implies that it should not respond to requests on any others.

Because the `nrpe` server is explicitly designed to run commands at the request of remote servers, it's very important to take steps like this where appropriate to prevent attackers from exploiting the service.

## See also

- The *Monitoring local services on a remote machine with NRPE* recipe in this chapter



# Setting allowed client hosts for NRPE

In this recipe, we'll learn how to configure the `nrpe` daemon to answer requests from a particular IP address, typically the designated Nagios Core server or servers monitoring your network. This means that `nrpe` will not run plugins or return results for any `check_nrpe` request made from IP addresses not in this list.

This is an elementary security step in running an NRPE server, as, if your target host has interfaces or routes into untrusted networks, there is a risk of attackers making spurious requests for information about the system, clogging up your disk with logs from excessive check requests, or even possibly exploiting the `nrpe` daemon or the Nagios Plugins. This should be done in concert with a hardware or software firewall and security policy.

# Getting ready

You should have a target host configured for checking in a Nagios Core 4.0 or later monitoring server. The target host should be running the nrpe daemon. You can verify that nrpe is running with `pgrep(1)` or `ps(1)`:

```
# pgrep nrpe
29964
# ps -e | grep [n]rpe
nagios 29964 1 0 21:55 ? 00:00:01 nrpe
```

We can verify that the target host is not configured to respond to a particular IP address by attempting to open a `telnet(1)` or `netcat(1)` connection to it. If we are not one of the allowed hosts, nrpe will close the session immediately without waiting for input:

```
$ telnet roma.example.net 5666
Trying 192.0.2.61...
Connected to 192.0.2.61.
Escape character is '^]'.
Connection closed by foreign host.
```

This assumes that NRPE is listening on its default port number 5666. In this example, we'll add the IP address `192.0.2.12` to the list of hosts allowed to request information from NRPE.

# How to do it...

We can configure the nrpe daemon to listen on a new address as follows:

1. Edit the nrpe daemon's configuration file. The default location is `/usr/local/nagios/etc/nrpe.cfg`. Look for the line beginning with `allowed_hosts`. It may look something like this:

```
allowed_hosts=127.0.0.1,192.0.2.11
```

2. Add or remove IP addresses from the line, separating them with commas. For this example, we're adding one more address:

```
allowed_hosts=127.0.0.1,192.0.2.11,192.0.2.12
```

3. Restart the nrpe daemon. If you have installed an `init` script for it, you may be able to this with something like:

```
# /etc/init.d/nrpe restart
```

If not, you can restart the process by sending it an HUP signal with the `pkill(1)` command, which will prompt it to re-read its configuration file and resume running:

```
# pkill -HUP nrpe
```

With this done, the nrpe daemon should now respond only to the nominated hosts making `check_nrpe` requests, immediately closing the connection otherwise. We can verify our new host is allowed to talk to the nrpe service on `roma.example.net` with another `telnet(1)` or `netcat(1)` test:

```
$ telnet roma.example.net 5666
```

Note that the nrpe daemon is now waiting for input rather than closing the connection immediately as it was doing before. This implies that we can now run `check_nrpe` checks from `192.0.2.12` if we need to.

## How it works...

The configuration we adjusted here defines a set of addresses to which the `nrpe` daemon should respond if a request is made and implies that it should refuse to answer requests made by any other address.

The `nrpe` daemon inspects the IP address of incoming connections and, if the `allowed_hosts` directive is defined, checks that the address features in that list. If it does not, it closes the connection and refuses to run any plugins, generating no output.



## There's more...

The `allowed_hosts` directive is actually optional; if we wished, we could set the `nrpe` server up to respond to requests from any IP address. The default installation and example configuration, however, enables it by default, allowing both requests from the `localhost` IP `127.0.0.1` and any network addresses the host had at the time `./configure` was run.

This is a sensible policy, because Nagios Core plugins are designed by third parties in an open-source community for monitoring purposes in trusted networks and may not necessarily be very secure. A plugin doesn't actually have to have a security hole to cause such problems; if the check it makes is very resource-intensive, for example opening a lot of TCP connections or querying a large database, an attacker could cause problems on the target host if allowed simply by making a large number of such `check_nrpe` requests in a short period of time.

## See also

- The *Monitoring local services on a remote machine with NRPE* recipe in this chapter



# Creating new NRPE command definitions securely

In this recipe, we'll learn how to securely create new command definitions for nrpe to run upon request by a monitoring server. We need to do this, because even if we have a huge set of plugins installed on our target host running nrpe, the daemon will only run commands defined in its configuration file.

We'll also learn how arguments can be passed to these commands, if strictly necessary, and about the potentially negative security consequences of this.

# Getting ready

You should have a target host configured for checking in a Nagios Core 4.0 or later monitoring server. The target host should be running the `nrpe` daemon. You can verify that `nrpe` is running with `pgrep(1)` or `ps(1)`:

```
# pgrep nrpe
29964
# ps -e | grep [n]rpe
nagios 29964 1 0 21:55 ? 00:00:01 nrpe
```

We can inspect the list of commands that `nrpe` is already configured to run by looking for command directives in its configuration file. By default, this file is `/usr/local/nagios/etc/nrpe.cfg`, and the default command definitions are near the end of the file:

```
command[check_users]=/usr/local/nagios/libexec/check_users -w 5 -c 10
command[check_load]=/usr/local/nagios/libexec/check_load -w 15,10,5 -c
30,25,20
command[check_hda1]=/usr/local/nagios/libexec/check_disk -w 20% -c 10% -p
/dev/hda1
command[check_zombie_procs]=/usr/local/nagios/libexec/check_procs -w 5 -c
10 -s Z
command[check_total_procs]=/usr/local/nagios/libexec/check_procs -w 150 -c
200
```

We'll add another command to this set to check if the swap space available is above a specific threshold using the standard Nagios Core plugin `check_swap`. We can test whether it's working first by running it on the target host:

```
$ /usr/local/nagios/libexec/check_swap -w 10% -c 5%
SWAP OK - 100% free (216 MB out of 217 MB) |swap=216MB;21;10;0;217
```

For completeness' sake, we'll also show how to define a service check using this new plugin on the Nagios Core monitoring server.

# How to do it...

We can add a new command definition to an nrpe configuration like as follows:

1. Edit the nrpe daemon's configuration file. The default location is `/usr/local/nagios/etc/nrpe.cfg`. Look for lines beginning with `command`, which are near the end of the file by default.
2. Add the following line to the end of the file:

```
command[check_swap]=/usr/local/nagios/libexec/check_swap -w 10% -c 5%
```

3. Restart the nrpe daemon. If you have installed an init script for it, you may be able to do this with something like:

```
# /etc/init.d/nrpe restart
```

If not, you can restart the process by sending it a HUP signal with the `kill(1)` command, which will prompt it to re-read its configuration file and resume running:

```
# kill -HUP nrpe
```

With this done, assuming that our monitoring server is part of the `allowed_hosts` directive and can contact the target host, a call to `check_nrpe` on the monitoring host should return the status and output of the `check_swap` plugin on the target host:

```
$ /usr/local/nagios/libexec/check_nrpe -H roma.example.net -c check_swap
SWAP OK - 100% free (216 MB out of 217 MB) |swap=216MB;21;10;0;217
```

In turn, this allows us to use the check in a service definition on the monitoring server with a `check_nrpe` command:

```
define service {
    use                generic-service
    host_name          roma.example.net
    service_description SWAP
    check_command       check_nrpe!check_swap
}
```

## How it works...

The preceding configuration defines a new command in `nrpe.cfg` called `check_swap`. The definition of new commands in `nrpe.cfg` takes the following general form:

```
command[command_name] = command_line
```

We defined a command `check_swap` for NRPE. It doesn't accept any arguments where the actual `check_swap` plugin requires them; instead, the arguments are hard-coded into the command definition, with the two options `-w 10%` and `-c 5%` setting the thresholds for free swap space.

Aside from checking system properties like the load average or the swap space, which might not otherwise be directly retrievable except via systems like SNMP, another common use for NRPE is to report on the state of network services that only listen locally, or are otherwise unreachable by the monitoring host. Database server monitoring is a good example. We could define the following command in `nrpe.cfg`:

```
command[check_mysql] = /usr/local/nagios/libexec/check_mysql -u nagios -d nagios -p NFmxenQ5
```

Assuming that the `check_mysql` plugin is installed on this target host, which it ought to be if the MySQL client library and headers were available at compile time, this would then enable this check to be run from the monitoring host:

```
$ /usr/local/nagios/libexec/check_nrpe -H roma.example.net -c check_mysql
Uptime: 420865  Threads: 1  Questions: 172  Slow queries: 0  Opens: 99
Flush tables: 1  Open tables: 23  Queries per second avg: 0.0
```

This can be configured as a service check using an appropriate command definition for `check_nrpe` as follows:

```
define service {
    use                generic-service
    host_name          roma.example.net
    service_description MySQL
    check_command       check_nrpe!check_mysql
}
```

Thus, we're able to get the status of the MySQL server on the remote host, `roma.example.net`, without actually connecting directly to the MySQL server itself; we arrange for the NRPE service on the target host to do it on the monitoring server's behalf.

This is not just useful for network services running on the target host. It can be used to delegate any kind of check that a remote host can perform where the monitoring server cannot. Using NRPE is, thus, also a way to work around the addressability problems of NAT; although a service running on an address like `192.168.1.1` would not be addressable from outside the network, if NRPE were running on the NAT gateway, we could use that to address the appropriate systems by their local addresses.

## There's more...

Moreover, near the bottom of `nrpe.cfg`, you'll find some information on providing arguments to NRPE commands as part of the `check_nrpe` request as opposed to hard-coding them. The comment included makes it quite clear that this carries some risks:

*The following examples allow user-supplied arguments and can only be used if the NRPE daemon was compiled with support for command arguments AND the `dont_blame_nrpe` directive in this config file is set to 1. This poses a potential security risk, so make sure you read the `SECURITY` file before doing this.*

It's important to understand that, when running NRPE on a target host, you are running a service that is designed to allow network machines to run commands on the target machine with no strong authentication, which is why keeping NRPE secure is so important. If you allow passing arguments to commands, you need to be aware of the full ramifications and risk of doing so and the recommended `SECURITY` file explains these well.

If you really want to use it, it requires reconfiguring and recompiling the `nrpe` daemon with the `--enable-command-args` switch:

```
$ ./configure --enable-command-args
$ make all
# make install-daemon
```

Then set the parameter `dont_blame_nrpe` in `nrpe.cfg` to 1, where it otherwise defaults to 0:

```
dont_blame_nrpe=1
```

After restarting `nrpe` (if you have rebuilt it, you will need to restart it completely this time and not simply HUP it), this allows us to use command definitions like the following:

```
command[check_mysql_args]=/usr/local/nagios/libexec/check_mysql -H
localhost -u $ARG1$ -d $ARG2$ -p $ARG3$
```

This, in turn, allows checks from the monitoring servers like the following, using the `check_nrpe` command's `-a` option:

```
# /usr/local/nagios/libexec/check_nrpe -H roma.example.net -c
check_mysql_args -a nagios nagios NFmxenQ5
```

Because of the security concerns, I would recommend you avoid using command arguments if at all possible. If you do absolutely need to use them, it's also important to make sure that the traffic is encrypted, especially if it contains usernames and passwords. Advice on how to manage this is included in the `SECURITY` document for the `nrpe` daemon.



## See also

- The *Monitoring local services on a remote machine with NRPE* recipe in this chapter
- The *Giving limited sudo(8) privileges to NRPE* recipe in this chapter
- Monitoring database services, [Chapter 5](#), *Monitoring Methods*



# Giving limited sudo(8) privileges to NRPE

In this recipe, we'll learn how to deal with the difficulty of executing permissions for NRPE. The majority of standard Nagios plugins don't require special privileges to run, although this also depends on how stringent your system's security restrictions are. However, some of the plugins require being run as root or perhaps as another user other than nagios. This is sometimes the case with plugins that need to make requests of system-level resources such as checking the integrity of RAID arrays.

There are four general approaches to fixing this:

- **Bad:** Change the plugins to `setuid`, meaning that they will always be run as the user who owns them, no matter who executes them. The problem with this is that setting this bit allows anyone to run the program as root, not just `nrpe`, a very common vector for exploits.
- **Worse:** Run `nrpe` as root or as the appropriate user. This is done by changing the `nrpe_user` and `nrpe_group` properties in `nrpe.cfg`. This is even more dangerous and completely inconsistent with the principle of least privilege—we should confer a user as little permission as possible to allow it to do its job. Never do this!
- **Better:** Use `command_prefix` in `nrpe.cfg` to prepend `/usr/bin/sudo` to all commands and give `nrpe` full `sudo(8)` privileges to run only the plugins in `/usr/local/nagios/libexec`. This is a bit better, but still quite risky as we likely don't need every single command to be run as root, only one or two.
- **Best:** Use `sudo(8)` to allow the `nrpe` user limited privileges for a subset of commands, only the ones it needs to run, and only as the user by which it needs to be run.

The last solution is the most likely to be secure, so we'll examine an example here. We'll run the plugin `check_procs` as root, to get a process count. In most cases, you wouldn't need root privileges to get a complete count of all processes, but it might be needed on a system with a very locked-down `grsecurity` patch installed, as an example.

# Getting ready

You should have a target host configured for checking in a Nagios Core 4.0 or later monitoring server. The target host should be running the nrpe daemon and listening on all interfaces. You can verify that nrpe is running with `pgrep(1)` or `ps(1)`:

```
# pgrep nrpe
29964
# ps -e | grep [n]rpe
nagios 29964 1 0 21:55 ? 00:00:01 nrpe
```

You should also have `sudo(8)` installed and working on the target system and understand what it does. We'll be editing the `/etc/sudoers` file to confer root privileges to our nrpe user for one program only. This recipe will assume that the nrpe daemon is running as the nagios user in the nagios group.

# How to do it...

We can confer limited root privileges for one command to our `nrpe` user as follows:

1. Edit the `/etc/sudoers` file. The safest way to do this is generally with a call to `visudo(8)`, which will make a temporary copy of the file and check that it makes sense before installing it:

```
# visudo
```

2. Add the following line to the file and save it:

```
nagios ALL=(ALL:ALL) NOPASSWD: /usr/local/nagios/libexec/check_procs
```

Note that if the `requiretty` directive appears anywhere in your `/etc/sudoers` file, you may need to remove it to make this work.

3. Become the `nagios` user with `sudo(8)` and test run the command works as root, with no password prompt:

```
# sudo -s -u nagios
$ sudo /usr/local/nagios/libexec/check_procs
PROCS OK: 89 processes
```

4. Edit the `nrpe` daemon's configuration file. The default location is `/usr/local/nagios/etc/nrpe.cfg`. Look for the command definition for `check_total_procs`; if there isn't one, create it. Note that `/usr/bin/sudo` has been added to the start of the command:

```
command[check_total_procs]=/usr/bin/sudo
/usr/local/nagios/libexec/check_procs -w 150 -c 200
```

5. Restart the `nrpe` daemon. If you have installed an `init` script for it, you may be able to do this with something like:

```
# /etc/init.d/nrpe restart
```

If not, you can restart the process by sending it a `HUP` signal with the `kill(1)` command, which will prompt it to re-read its configuration file and resume running:

```
# kill -HUP nrpe
```

With this done, we should now be able to run a `check_nrpe` call from the monitoring server and get a successful response:

```
$ /usr/local/nagios/libexec/check_nrpe -H roma.example.net -c
check_total_procs
PROCS OK: 89 processes
```

## How it works...

The preceding configuration does not change nrpe daemon's behavior very much; most of the configuration is actually done on its host system. All we changed was the command definition for `check_total_procs` to run it from within `sudo(8)`.

To make this work without a password, we defined it in the `/etc/sudoers` file so that no password was required to execute this particular program as root for the nagios user, which is the user that nrpe runs.

This means that when we call the `check_total_procs` command from the monitoring server, it returns us the full output of the plugin as it was run with root privileges, but the nagios user doesn't have root privileges to run anything else potentially dangerous, such as `rm(1)` or `halt(8)`.

## There's more...

While this is a much more secure way of allowing privileges as another user for nrpe, it still requires trusting that the plugin that is being run with root privileges is secure and can't easily be exploited. Be very careful running this with custom code or with stray plugins you find on the web!

If you intend to allow the nagios user to run more than a couple of distinct programs, it may look a little tidier to define them in `/etc/sudoers` with a `Cmnd_Alias`:

```
Cmnd_Alias NAGIOS = /usr/local/nagios/libexec/check_procs,  
/usr/local/nagios/libexec/check_load  
nagios ALL=(ALL) NOPASSWD: NAGIOS
```

## See also

- The *Monitoring local services on a remote machine with NRPE* recipe in this chapter
- The *Using check\_by\_ssh with key authentication instead of NRPE* recipe in this chapter





# Using `check_by_ssh` with key authentication instead of NRPE

While all of the previous recipes in this chapter show that NRPE can be very effectively tied down and secured, it may be that we require some means of authentication to a target host in order to run the appropriate Nagios plugins on it. The `nrpe` daemon does not require any authentication to return information about the host's state; as long as the IP addresses all match and the command is defined for running, it will return information.

If you already use SSH keys for a public key infrastructure in your network, then you may find it preferable to use the `check_by_ssh` plugin instead, which allows you to use public keys to authenticate with a target host before running any commands. This is only suitable if the target host runs a secure shell daemon.

In this recipe, we'll repeat the setup for the `check_load` plugin as done in the first recipe in this chapter, *Monitoring local services on a remote machine with NRPE*, but we'll use the `check_by_ssh` plugin instead.

# Getting ready

You should have a Nagios Core 4.0 or newer server ready with the `ssh(1)` client software installed and a target host running an `sshd(8)`. The widely-deployed and open source OpenSSH implementations should work fine.

The target host should have all the Nagios Plugins installed, in this case including `check_load`.

You should have some familiarity with public key authentication and its advantages and disadvantages. Wikipedia has an excellent article about public key cryptography and authentication here: [http://en.wikipedia.org/wiki/Public-key\\_cryptography](http://en.wikipedia.org/wiki/Public-key_cryptography)

There is a popular introduction to SSH authentication with a key infrastructure here: [http://support.suso.com/supki/SSH\\_Tutorial\\_for\\_Linux](http://support.suso.com/supki/SSH_Tutorial_for_Linux)

We will be running through a sensible key infrastructure setup, but it would pay to have an understanding of how it works in case this setup does not suit your configuration.

# How to do it...

We can arrange to get the output of `check_load` from our remote host by way of `check_by_ssh` as follows:

1. On the monitoring server, decide on a location for the private and public keys for the nagios user. A sensible location is in the Nagios user's home directory, in this example, `/home/nagios`.
2. Become the nagios user using `su(8)` or `sudo(8)`:

```
# su nagios
# sudo -s -u nagios
```

3. Generate a pair of private and public SSH keys using `ssh-keygen`.

```
$ ssh-keygen
```

When prompted for a passphrase, simply press *Enter* to signal that you don't want one. You will have to do this twice.

This should create two files in `/home/nagios/.ssh`, called `id_rsa` and `id_rsa.pub`. The first one is the **private key** and should be kept secret at all times. The second, the **public key**, is safe to distribute and to install on other machines. The `rsa` portion of the filename may be different depending on which algorithm your system used to generate the keys; it may be `ecdsa`, for example.

4. On the target host, decide on a location for the `authorized_keys` file. The easiest way to do this is probably by specifying a home directory for the nagios user and creating it if appropriate:

```
# nagioshome=/home/nagios
# usermod -d "$nagioshome" nagios
# mkdir -p "$nagioshome"/.ssh
# chown -R nagios.nagios "$nagioshome"
# chmod 0700 "$nagioshome"/.ssh
```

5. Copy the file `id_rsa.pub` from the previous step from the monitoring server to `/home/nagios/.ssh/authorized_keys` on the target machine. The best way to do this varies. One possible method is to use `scp(1)`:

```
$ whoami
nagios
$ scp /home/nagios/.ssh/id_rsa.pub
roma.example.net:/.ssh/authorized_keys
```

You may have to set a password temporarily for the nagios user on the target host to do this:

```
# passwd nagios
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully.
```

Check that you can now log in from the monitoring server to the target server as the

nagios user with no password:

```
$ whoami
nagios
$ ssh roma.example.net
Linux roma 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686
Lost login: Sun Jul 29 18:23:14 2012 from olympus.example.net
```

6. Back on the monitoring server, check that you're able to run the `check_by_ssh` plugin to call one of the installed plugins on the remote server, in this example, `check_load`:

```
$ whoami
nagios
$ /usr/local/nagios/libexec/check_by_ssh \
    -H roma.example.net \

    -C '/usr/local/nagios/libexec/check_load -w 15,10,5 -c 30,24,20'
OK - load average: 0.00, 0.00, 0.00|load1=0.000;15.000;30.000;0;
load5=0.000;10.000;24.000;0; load15=0.000;5.000;20.000;0;
```

Note that the value for the `-c` option needs to be surrounded with quotes.

7. Now that we've verified that `check_by_ssh` works with our infrastructure, we can define a command to use it in `/usr/local/nagios/etc/objects/commands.cfg`:

```
define command {
    command_name    check_by_ssh
    command_line    $USER1$/check_by_ssh -H $HOSTADDRESS$ -i
/home/nagios/.ssh/id_rsa -C '$ARG1$'
}
```

8. We can apply that command to a new service check for `roma.example.net` as follows, preferably placed beneath the host definition:

```
define service {
    use                generic-service
    host_name          roma.example.net
    service_description LOAD_BY_SSH
    check_command
check_by_ssh!/usr/local/nagios/libexec/check_load -w 15,10,5 -c
30,25,20
}
```

9. Validate the configuration, and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios reload
```

With this done, a new service with the description `LOAD_BY_SSH` should appear in the web interface ready to be checked and will come up with an appropriate status, including the load average as read via SSH to the target host. To Nagios Core, the results of the check are just the same as if they'd come via NRPE.

# How it works...

Checking services via NRPE and SSH is actually a reasonably similar process; the central idea is that we're using another network service shared by the monitoring server and the target host to instruct the target host to run a plugin and return the results to the monitoring server.

In the case of `check_by_ssh`, this is done over an SSH connection. It's quite typical for administrators to run commands on remote hosts using the `ssh(1)` client simply by adding the command to be run to the command line:

```
$ hostname  
olympus.example.net  
$ ssh roma.example.net hostname  
roma.example.net
```

The `check_by_ssh` plugin does this in a Nagios plugin context. The value of the `command_line` directive for the command we defined can be broken down as follows:

- `$USER1$/check_by_ssh`: The path to the plugin, as typical in command definitions, using the `$USER1$` macro to expand to `/usr/local/nagios/libexec`.
- `-H $HOSTADDRESS$`: This command specifies that the plugin should connect to the applicable host for any host or service that uses this command.
- `-C '$ARG1$'`: This command specifies that the command run by `check_by_ssh` on the target host is given in the first argument that the service defines in its `check_command`; in this case, our command is:

```
/usr/local/nagios/libexec/check_load -w 15,10,5 -c 30,25,20
```

Note that it's important that `$ARG1$` is in quotes, because we want to pass the whole command as one argument to `check_by_ssh`.

## There's more...

Using `check_by_ssh` instead of `check_nrpe` allows full authentication via public key rather than merely by having the right IP address. It also encrypts all traffic implicitly, minimizing the risk of sensitive data like usernames or passwords being intercepted. Which of the two plugins to use will depend very much on the nature of your network and your security policy. There is no reason you can't use both of them if you wish.

Be careful not to confuse this plugin with `check_ssh`, which checks that the SSH service itself is running and is discussed in the *Monitoring SSH for any host* recipe in [Chapter 5, Monitoring Methods](#).

You may wish to configure the `ssh(1)` client on the monitoring server to disallow any use of password authentication and use only public keys, since the checks are automated processes that can't accept password input. An example `~/.ssh/config` file suitable for public-key authentication from Nagios might look like the following; check the manual page for `ssh_config(5)` to see more detail on each of these options.

```
# Don't allow password or passphrase input
BatchMode yes
# Don't use challenge-response authentication
ChallengeResponseAuthentication no
# Don't check the IP of the target host
CheckHostIP no
# Don't use an SSH Escape character
EscapeChar none
# Reduce output a bit; only print errors
LogLevel ERROR
# Don't try to authenticate with passwords
PasswordAuthentication no
# Don't check host keys to check we're authenticating with the right host;
use with caution
StrictHostKeyChecking no
```

## See also

- The *Monitoring local services on a remote machine with NRPE* recipe in this chapter
- *Monitoring SSH for any host*, [Chapter 5](#), *Monitoring Methods*





# Using check\_mk instead of NRPE

On very large networks, or networks with a lot of remote checking required, manually configuring NRPE services for everything that requires checking on each host can be burdensome. It's common to automate installing the plugins that are needed on the monitored host and the configuration required to use them, to make the network administrator's job a little easier when adding a set of new hosts with a standard set of services to monitor. Common tools to automate such setups include configuration management tools like Chef, Puppet, or Bcfg2.

Developing standard procedures like this may not appeal and busy administrators may be looking for an extension to Nagios that automatically discovers relevant services on a host to monitor, so that almost no manual configuration is required, instead generating the relevant stanzas automatically. This is one of the main functions of **Check\_MK**, a Nagios Core extension written by Matthias Kettner and available at [https://mathias-kettner.de/check\\_mk.html](https://mathias-kettner.de/check_mk.html).

Check\_MK is a very large Nagios Core extension with a great deal of functionality that makes it a large portion of a monitoring system in its own right, going so far as to have its own web GUI. Covering it in its entirety is beyond the scope of this book, but its particular function in automating and generating checks for remote hosts is relevant here.

# Getting ready

In this example, we'll set up the `check_mk` program on a server running Nagios Core 4.0 or better, `olympus.example.net`, and install the `check_mk_agent` on a second server, `roma.example.net`, so that checks can be made of the latter remotely, with automatically generated configuration.

There shouldn't already be a host named `roma` in the Nagios Core configuration; this would cause configuration errors. The host should be pingable by its hostname and its firewall, if any, should be configured to allow TCP port 6556 from the monitoring server.

The target server should also have `xinetd(8)` installed and running; the package is named `xinetd` in both Debian and Red Hat derived operating systems.

You will need to have root privileges on both the monitoring server and the target server. The `check_mk` program installs a great deal of software and modifies some system configuration, so it's not trivial to remove it and you may like to test this on a development network first.

The `check_mk` program is part of a large suite of monitoring tools that are downloaded together; you'll need to be able to download around 100 MB of data to the monitoring server.

# How to do it...

We can set up check\_mk on the monitoring server as follows:

1. Download the most recent version of the check\_mk suite's source; at the time of writing, the link is available at [http://mathias-kettner.com/check\\_mk\\_download\\_version.php](http://mathias-kettner.com/check_mk_download_version.php). It is a large file (100 MB).

```
$ wget https://mathias-kettner.de/support/1.2.6p9/check-mk-raw-1.2.6p9.cre.tar.gz
```

2. Unpack the downloaded tarball and move into the packages/check\_mk directory within it:

```
$ tar -xzf check-mk-raw-1.2.6p9.cre.tar.gz
$ cd check-mk-raw-1.2.6p9/packages/check_mk
```

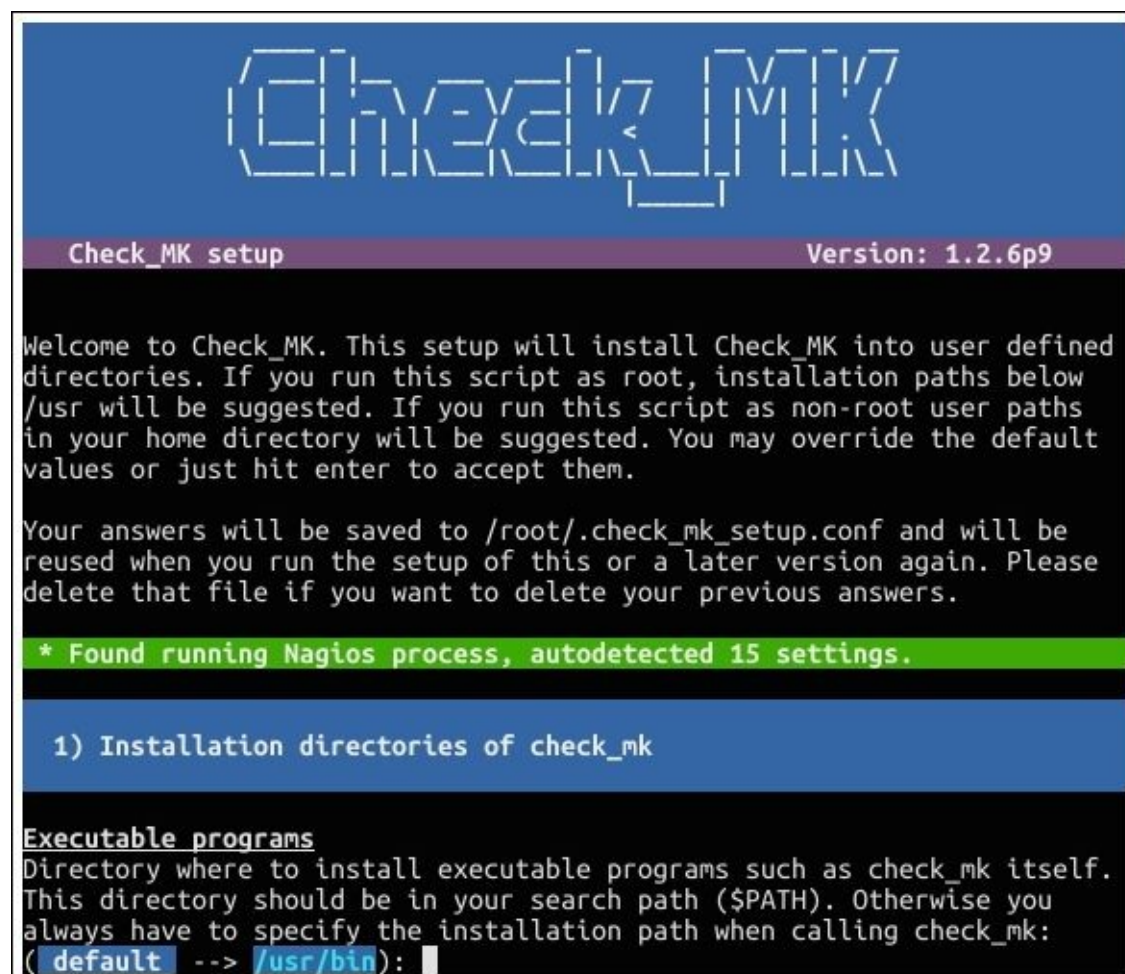
3. Unpack the check\_mk tarball in that directory and change into it:

```
$ tar -xzf check_mk-1.2.6p9
$ cd check_mk
```

4. Run the setup.sh script within the directory as the root user:

```
# ./setup.sh
```

The check\_mk installation script interface should appear:



```
Check_MK setup Version: 1.2.6p9

Welcome to Check_MK. This setup will install Check_MK into user defined
directories. If you run this script as root, installation paths below
/usr will be suggested. If you run this script as non-root user paths
in your home directory will be suggested. You may override the default
values or just hit enter to accept them.

Your answers will be saved to /root/.check_mk_setup.conf and will be
reused when you run the setup of this or a later version again. Please
delete that file if you want to delete your previous answers.

* Found running Nagios process, autodetected 15 settings.

1) Installation directories of check_mk

Executable programs
Directory where to install executable programs such as check_mk itself.
This directory should be in your search path ($PATH). Otherwise you
always have to specify the installation path when calling check_mk:
( default --> /usr/bin):
```

5. Answer the questions the setup program asks. In most cases, the default it suggests should be appropriate, as it attempts to read the configuration of your system and find running instances of Nagios Core.
6. Copy three files from the new directory `/usr/share/check_mk/agents` to the target host, perhaps by using `scp(1)`:

```
$ cd /usr/share/check_mk_agents
$ scp check_mk_agent.linux waitmax xinetd.conf roma:
```

7. On the target server, install the files copied from the monitoring server; note the different names given to `check_mk_agent.linux` and `xinetd.conf`:

```
# cp check_mk_agent.linux /usr/bin/check_mk_agent
# cp waitmax /usr/bin/waitmax
# cp xinetd.conf /etc/xinetd.d/check_mk
```

8. Restart the `xinetd(8)` server on the target machine:

```
# /etc/init.d/xinetd restart
```

9. Back on the monitoring server, edit the file `/etc/check_mk/main.mk` to include the hostname of your target host as the only value for `all_hosts`.

```
all_hosts = [ "roma" ]
```

10. Run `check_mk -I` to update the inventory of checks for all of Check\_MK's configured hosts.

```
# check_mk -I
```

11. Run `check_mk -O` to write new configuration to Nagios Core's configuration directory and restart it.

```
# check_mk -O
Generating configuration for core (type nagios)...OK
Validating Nagios configuration...OK
Precompiling host checks...OK
Reloading monitoring core...OK
```

With this done, if you visit the Nagios Core web administration area, you should see a host has been automatically configured for you along with a set of service checks. The checks you see will vary depending on the activity of your host, as shown in the following screenshot:

|      |  |                          |   |    |                     |              |     |                                                                                                                                                |
|------|--|--------------------------|---|----|---------------------|--------------|-----|------------------------------------------------------------------------------------------------------------------------------------------------|
| roma |  | CPU load                 | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - 15min load 0.05                                                                                                                           |
|      |  | CPU utilization          | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - user: 0.1%, system: 0.1%, wait: 0.2%, total: 0.5%                                                                                         |
|      |  | Check_MK                 |   | OK | 08-06-2015 21:21:54 | 0d 0h 1m 20s | 1/1 | OK - Agent version 1.2.6p9, execution time 0.2 sec                                                                                             |
|      |  | Disk IO SUMMARY          | ? | OK | 08-06-2015 21:21:55 | 0d 0h 0m 19s | 1/1 | OK - 0.00 B/sec read, 409.60 B/sec write, IOs: 0.10/sec                                                                                        |
|      |  | Interface 2              | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - [eth0] (up) speed unknown, in: 89.62 B/s, out: 243.72 B/s                                                                                 |
|      |  | Kernel Context Switches  | ? | OK | 08-06-2015 21:21:55 | 0d 0h 0m 19s | 1/1 | OK - 41/s                                                                                                                                      |
|      |  | Kernel Major Page Faults | ? | OK | 08-06-2015 21:21:55 | 0d 0h 0m 19s | 1/1 | OK - 0/s                                                                                                                                       |
|      |  | Kernel Process Creations | ? | OK | 08-06-2015 21:21:55 | 0d 0h 0m 19s | 1/1 | OK - 1/s                                                                                                                                       |
|      |  | Memory used              | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - 0.08 GB used (0.08 RAM + 0.00 SWAP + 0.00 Pagetables, this is 17.3% of 0.48 RAM (0.25 total SWAP)), 0.0 mapped, 0.5 committed, 0.0 shared |
|      |  | Mount options of /       | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - mount options exactly as expected                                                                                                         |
|      |  | Number of threads        | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - 136 threads                                                                                                                               |
|      |  | Postfix Queue            | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - The mailqueue is empty                                                                                                                    |
|      |  | TCP Connections          | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - ESTABLISHED: 2, TIME_WAIT: 1                                                                                                              |
|      |  | Uptime                   | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - up since Wed Aug 5 21:04:13 2015 (1d 00:17:41)                                                                                            |
|      |  | fs_/_                    | ? | OK | 08-06-2015 21:21:55 | 0d 0h 1m 20s | 1/1 | OK - 34.0% used (1.54 of 4.54 GB), (levels at 80.00/90.00%), trend: 0.00 B / 24 hours, inodes available 262k/84.18%                            |

## How it works...

We installed Check\_MK on the Nagios Core monitoring server, allowing its use both as a plugin from Nagios Core and as a tool to detect services to monitor on target systems and to generate configuration for Nagios Core to monitor them. We then edited its main configuration file to specify that we wanted to detect services and generate configuration for the remote roma host specifically.

We then copied some support files from the Check\_MK suite to the target host to install the agent, check\_mk\_agent, and the configuration for the internet superserver xinetd(8) to run it. The agent then starts listening on port 6556. You can test that this is working by connecting to the port with a tool like telnet(1) or netcat(1):

```
$ telnet roma 6556
Trying 192.0.2.61...
Connected to roma.example.net.
Escape character is '^]'.
<<<check_mk>>>
Version: 1.2.6p9
AgentOS: linux
AgentDirectory: /etc/check_mk
DataDirectory: /var/lib/check_mk_agent
SpoolDirectory: /var/lib/check_mk_agent/spool
...
```

This agent is the program that check\_mk on the monitoring server contacts to get all the details about the target system's services. It requests all the details in a single check and then updates the details for all the monitored services from that data, using passive checks. This means that, unlike check\_nrpe, check\_mk only needs to run once to get all the data for that host.

Once this was done, we ran check\_mk with its -I option, which updates its inventory of all the hosts it monitors and the services on them, and then with its -o option; this generates configuration for all the host and services checks into a new directory, in our case /usr/local/nagios/etc/check\_mk.d, saving us the trouble of writing the configuration ourselves. In our example, Check\_MK generated monitoring configuration for CPU load, CPU utilization, disk I/O, and many other services, all automatically. It even detected that we were running Postfix to monitor the number of messages in the mail processing queue!

## There's more...

Check\_MK's functionality is not limited to being a replacement for `check_nrpe` or `check_ssh`; its functionality goes far deeper than that and advanced usage of it could fill a book in itself. If you think the automation that Check\_MK offers is the way you'd like to configure Nagios Core, you can get a lot more information about Check\_MK and the monitoring distribution in which it's included on its website at [https://mathias-kettner.com/check\\_mk.html](https://mathias-kettner.com/check_mk.html).

Similar security considerations apply to Check\_MK deployment as do NRPE or SSH deployment. By default, the `xinetd.conf` file included with the system allows any host to query the data. If the target server is open to the internet without a firewall, this would be a major security hole. You should restrict network access to the service so that only the monitoring server can access the TCP port 6556 on the target server.



## See also

- The *Monitoring local services on a remote machine with NRPE* recipe in this chapter
- The *Using check\_by\_ssh with key authentication instead of NRPE* recipe in this chapter
- *Submitting passive checks from a remote host with NSCA*, [Chapter 10](#), *Security and Performance*
- *Reading status from a UNIX socket with mklivestatus*, [Chapter 10](#), *Security and Performance*



# Chapter 7. Using the Web Interface

In this chapter, we will cover the following recipes:

- Using the Tactical Overview
- Viewing and interpreting availability reports
- Viewing and interpreting trends
- Viewing and interpreting notification history
- Adding comments on hosts or services in the web interface
- Viewing configuration in the web interface
- Scheduling checks from the web interface
- Acknowledging a problem via the web interface

# Introduction

The Nagios Core web interface is an administrator's first port of call to see the current status of the network being monitored. By way of CGI and PHP scripts, it allows both an overview of the general performance of all the hosts and services being monitored and, going into more detail, about their current states and how checks are being performed. This provides considerably more details than are normally contained in e-mailed notifications. The default home page for the Nagios Core web administration area contains process information, news, and quick links, as shown in the following screenshot:

The screenshot shows the Nagios Core web interface. At the top left is the 'Nagios' logo. Below it is a sidebar menu with sections: 'General' (Home, Documentation), 'Current Status' (Tactical Overview, Map (Legacy), Hosts, Services, Host Groups, Service Groups, Problems), 'Reports' (Availability, Trends (Legacy), Alerts, Notifications, Event Log), and 'System' (Comments, Downtime, Process Info, Performance Info, Scheduling Queue, Configuration). The main content area features the 'Nagios Core' logo with a green checkmark indicating 'Daemon running with PID 5714'. Below this, it shows 'Nagios Core Version 4.1.1' and the date 'August 19, 2015', with a link to 'Check for updates'. A large banner for the 'Nagios World Conference 2015' is displayed, with details: 'St. Paul, MN | Sept. 28-30', 'Learn. Experience. Implement.', and a 'Register Today >>' button. Below the banner are four sections: 'Get Started' (Start monitoring your infrastructure, Change the look and feel of Nagios, Extend Nagios with hundreds of addons, Get support, Get training, Get certified), 'Quick Links' (Nagios Library, Nagios Labs, Nagios Exchange, Nagios Support, Nagios.com, Nagios.org), 'Latest News' (Nagios Plugins 2.0.2 Released, Nagios Projects Moved To GitHub, Nagios Core 4.0.6 Released, More news...), and 'Don't Miss...' (Improve your Nagios skillset, Use Nagios to Check if You're Vulnerable to the FREAK Security Vulnerability, The first release candidate for Nagios Core 4.1.0 has been released).

In most respects, the Nagios Core web interface (without any add-ons) is mostly geared for the display of information rather than configuring the server, but there are some things that can be done in it to actually change the way Nagios Core runs. These include the following tasks:

- Disabling or enabling active and passive checks, event handlers, notifications, and flap detection, whether for all hosts and services or specific ones
- Sending custom notifications about a host or service that don't necessarily correspond to an actual state change
- Rescheduling checks, which is commonly done to prioritize a check when there have

been problems that the administrator thinks are now resolved and wishes to recheck

- Acknowledging problems, which is done to allow administrators to suppress further notifications from problems that they're working to fix
- Scheduling downtime to suppress notifications for a certain time period; this was discussed in the *Scheduling downtime for a host or service* recipe, in [Chapter 3](#), *Working with Checks and States*
- Leaving comments on hosts and services for the benefit of any other administrator who inspects them; many of the actions in this list will do this implicitly

We won't offer a comprehensive survey of the web interface here, as much of its use is broken down in other chapters or is otherwise reasonably self-explanatory. Instead, we'll explore the few runtime behavior changes we can arrange through the web interface listed before and on using and interpreting the reports available under the **Reports** heading of the left-hand side navigation menu.

Another important and extremely useful part of the web interface, the Network Status Map, is discussed in several recipes in [Chapter 8](#), *Managing Network Layout*. We won't discuss it here, as it's so useful that it merits half a chapter in itself.



# Using the Tactical Overview

In this recipe, we'll take a look at the **Tactical Overview** screen of the Nagios Core web interface. Like its name implies, the screenshot on the following page provides a one-page summary of the current operating status of both the monitored hosts and services and the Nagios Core server itself.

# Getting started

You will need access to the Nagios Core web interface. In the default installation, the nagiosadmin user has the necessary privileges.



# How to do it...

We can take a look at the Tactical Overview as follows:

1. Log in to the Nagios Core web interface.
2. Click on the **Tactical Overview** item in the left-hand side menu:



You should see the **Tactical Monitoring Overview** screen appear in the right-hand side frame:

**Nagios®**

**Tactical Monitoring Overview**  
Last Updated: Sun Sep 13 16:46:15 NZST 2015  
Updated every 90 seconds  
Nagios® Core™ 4.1.1 - www.nagios.org  
Logged in as nagiosadmin

**Monitoring Performance**

|                                  |                         |
|----------------------------------|-------------------------|
| Service Check Execution Time:    | 0.00 / 4.01 / 0.796 sec |
| Service Check Latency:           | 0.00 / 0.00 / 0.000 sec |
| Host Check Execution Time:       | 3.00 / 4.01 / 3.502 sec |
| Host Check Latency:              | 0.00 / 0.00 / 0.000 sec |
| # Active Host / Service Checks:  | 2 / 9                   |
| # Passive Host / Service Checks: | 0 / 14                  |

**Network Health**

Host Health:

Service Health:

**General**

- Home
- Documentation

**Current Status**

- Tactical Overview**
- Map (Legacy)
- Hosts
- Services
- Host Groups
  - Summary
  - Grid
- Service Groups
  - Summary
  - Grid
- Problems
  - Services (Unhandled)
  - Hosts (Unhandled)
  - Network Outages

Quick Search:

**Reports**

- Availability
- Trends (Legacy)
- Alerts
  - History
  - Summary
  - Histogram (Legacy)
- Notifications
- Event Log

**System**

- Comments
- Downtime
- Process Info
- Performance Info
- Scheduling Queue
- Configuration

**Network Outages**

0 Outages

**Hosts**

1 Down    0 Unreachable    1 Up    0 Pending

1 Unhandled Problems

**Services**

1 Critical    0 Warning    0 Unknown    22 Ok    0 Pending

1 on Problem Hosts    14 Disabled

**Monitoring Features**

| Flap Detection         | Notifications         | Event Handlers         | Active Checks          | Passive Checks         |
|------------------------|-----------------------|------------------------|------------------------|------------------------|
| ✓ All Services Enabled | ✓ 2 Services Disabled | ✓ 15 Services Disabled | ✓ 14 Services Disabled | ✓ All Services Enabled |
| No Services Flapping   | All Hosts Enabled     | 1 Host Disabled        | All Hosts Enabled      | All Hosts Enabled      |
| All Hosts Enabled      |                       |                        |                        |                        |
| No Hosts Flapping      |                       |                        |                        |                        |

3. Try clicking on some of the items under **Hosts** and **Services**, for example, the **Up** count under **Hosts**. Note that you're presented with a listing of all the hosts that comprise that section.
4. Return to the **Tactical Overview**, and try clicking on some of the items under the **Monitoring Features** heading, for example, the **Enabled** item under **Flap Detection**. Note that clicking on these brings you to a screen where you can quickly turn the relevant feature on or off (provided you have the appropriate privileges).

## How it works...

The Tactical Overview is a good first port of call whenever you become aware of problems with the monitored network, particularly, if you've a reason to believe that the problems are on more than one host. It's also a good place to visit to adjust monitoring features. For example, if a very large segment of your network is going to be down and it's not practical to schedule downtime for all the affected hosts, you could simply suppress notifications completely for a while in order to fix the problem with no distractions.

Note that the screen includes an overall assessment of the monitoring performance of the server in the top-right corner; this can be a useful first place to look if you suspect that the monitoring server is struggling to keep up with its tasks or want to get some idea of how many checks it's conducting.

Below that is a simple bar graph showing the general state of the network. It will change color depending on the severity of the problems; in the preceding screenshot, there are serious problems with some of the hosts on this network, and as a result, the bar is short and red in color.

## There's more...

If you find the Tactical Overview useful, it may be a good idea to make it the home page for the web interface, rather than the default page, which mostly just shows the Nagios Core branding, version, and some links. We can arrange this by changing the value of `$url` in `/usr/local/nagios/share/index.php` from `main.php` to `cgi-bin/tac.cgi`:

```
$url = 'cgi-bin/tac.cgi';
```

With this done, when we visit the `/nagios/` URL on the monitoring host, the Tactical Overview should come up immediately. This can be done with any of the pages in the menu; the **Services** item is another good choice.

## See also

- The *Viewing configuration in the web interface* recipe in this chapter



# Viewing and interpreting availability reports

In this recipe, you'll learn how to use the **Availability Report** to build a table showing statistics for a host, hostgroup, service, or servicegroup. This is useful as a quick metric of overall availability, perhaps to meet the terms of a service-level agreement.

# Getting started

You will need access to the Nagios Core web interface and permission to run commands from the CGIs. The sample configuration installed by following the Quick Start Guide gives the nagiosadmin user all the necessary privileges when authenticated via HTTP.

If you find that you don't have this privilege, check the `authorized_for_all_services` and `authorized_for_all_hosts` directives in `/usr/local/nagios/etc/cgi.cfg` and include your username in both, for example tom:

```
authorized_for_all_services=nagiosadmin,tom
authorized_for_all_hosts=nagiosadmin,tom
```

Alternatively, you should also be able to see a host or service's information if you are authenticating with the same username as the nominated contact for the host or service you want to check. This is explained in the *Using authenticated contacts* recipe in [Chapter 10, Security and Performance](#).

In this example, we'll view a month's history for a CPU load service on `roma.example.net`, a Linux server.

# How to do it...

We can arrange an **Availability Report** for the last month for our `roma.example.net` server like so:

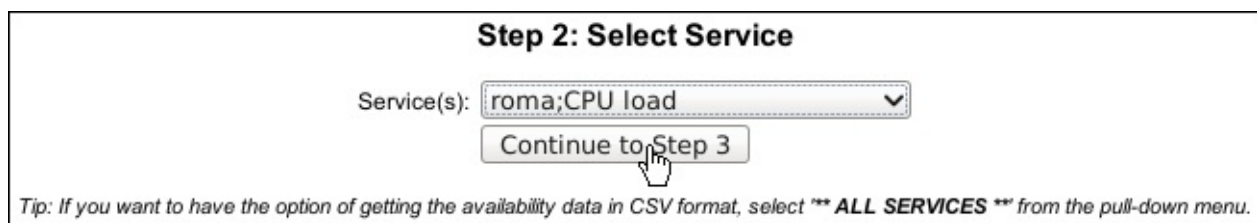
1. Log in to the Nagios Core web interface.
2. Click on the **Availability** link in the left-hand side menu, beneath **Reports**:



3. Select the type of report, either **Host**, **Hostgroup**, **Service**, or **Servicegroup**:



4. Select a specific host or service on which to report or alternatively choose to run it for all hosts, hostgroups, services, or servicegroups:



5. Define some options for the report. The defaults are sensible, so we'll use them for now; we'll go over the effect of each of the other fields in the next section. Click on **Create Availability Report!** when you're done:



### Step 3: Select Report Options

Report Period: Last 7 Days ▼

If Custom Report Period...

Start Date (Inclusive): September ▼ 1 2015

End Date (Inclusive): September ▼ 13 2015

Report time Period: None ▼

Assume Initial States: Yes ▼

Assume State Retention: Yes ▼

Assume States During Program Downtime: Yes ▼

Include Soft States: No ▼

First Assumed Service State: Unspecified ▼

Backtracked Archives (To Scan For Initial States): 4

Create Availability Report!

You should be presented with a table showing the percentage of time the host or service has spent in each state. A healthy service might look like this, with a few blips or none at all:

| Service State Breakdowns: |                    |               |              |              |
|---------------------------|--------------------|---------------|--------------|--------------|
| State                     | Type / Reason      | Time          | % Total Time | % Known Time |
| OK                        | Unscheduled        | 6d 22h 10m 0s | 98.909%      | 98.909%      |
|                           | Scheduled          | 0d 0h 0m 0s   | 0.000%       | 0.000%       |
|                           | Total              | 6d 22h 10m 0s | 98.909%      | 98.909%      |
| WARNING                   | Unscheduled        | 0d 1h 45m 0s  | 1.042%       | 1.042%       |
|                           | Scheduled          | 0d 0h 0m 0s   | 0.000%       | 0.000%       |
|                           | Total              | 0d 1h 45m 0s  | 1.042%       | 1.042%       |
| UNKNOWN                   | Unscheduled        | 0d 0h 0m 0s   | 0.000%       | 0.000%       |
|                           | Scheduled          | 0d 0h 0m 0s   | 0.000%       | 0.000%       |
|                           | Total              | 0d 0h 0m 0s   | 0.000%       | 0.000%       |
| CRITICAL                  | Unscheduled        | 0d 0h 5m 0s   | 0.050%       | 0.050%       |
|                           | Scheduled          | 0d 0h 0m 0s   | 0.000%       | 0.000%       |
|                           | Total              | 0d 0h 5m 0s   | 0.050%       | 0.050%       |
| Undetermined              | Nagios Not Running | 0d 0h 0m 0s   | 0.000%       |              |
|                           | Insufficient Data  | 0d 0h 0m 0s   | 0.000%       |              |
|                           | Total              | 0d 0h 0m 0s   | 0.000%       |              |
| All                       | Total              | 7d 0h 0m 0s   | 100.000%     | 100.000%     |

A more problematic service might have large percentages of time in the WARNING or CRITICAL state:

| Service State Breakdowns: |                    |                |              |              |
|---------------------------|--------------------|----------------|--------------|--------------|
| State                     | Type / Reason      | Time           | % Total Time | % Known Time |
| OK                        | Unscheduled        | 15d 5h 36m 51s | 49.142%      | 49.142%      |
|                           | Scheduled          | 0d 0h 0m 0s    | 0.000%       | 0.000%       |
|                           | Total              | 15d 5h 36m 51s | 49.142%      | 49.142%      |
| WARNING                   | Unscheduled        | 0d 0h 0m 0s    | 0.000%       | 0.000%       |
|                           | Scheduled          | 0d 0h 0m 0s    | 0.000%       | 0.000%       |
|                           | Total              | 0d 0h 0m 0s    | 0.000%       | 0.000%       |
| UNKNOWN                   | Unscheduled        | 0d 0h 0m 0s    | 0.000%       | 0.000%       |
|                           | Scheduled          | 0d 0h 0m 0s    | 0.000%       | 0.000%       |
|                           | Total              | 0d 0h 0m 0s    | 0.000%       | 0.000%       |
| CRITICAL                  | Unscheduled        | 3d 17h 15m 0s  | 11.996%      | 11.996%      |
|                           | Scheduled          | 12d 1h 8m 9s   | 38.862%      | 38.862%      |
|                           | Total              | 15d 18h 23m 9s | 50.858%      | 50.858%      |
| Undetermined              | Nagios Not Running | 0d 0h 0m 0s    | 0.000%       |              |
|                           | Insufficient Data  | 0d 0h 0m 0s    | 0.000%       |              |
|                           | Total              | 0d 0h 0m 0s    | 0.000%       |              |
| All                       | Total              | 31d 0h 0m 0s   | 100.000%     | 100.000%     |

The preceding table appears and gives a quick visual summary of the time spent in each state, which if clicked, will link to a **Trends Report** with the same criteria. Additionally, below the table there are log entries for the service or host that is changing its state.

# How it works...

Nagios Core assembles state changes from its log files for the specified time period and constructs the table of state changes by percentage. The availability report, therefore, only works for the time covered by your archived log files. The third step of building the report involves a lot of possible options, which are as follows:

- **Report period:** This is the drop-down menu that allows us to choose a fixed period, for convenience, relative to the current date; alternatively, a custom time period may be used by selecting the final **CUSTOM TIME PERIOD** option and selecting dates in the two fields that follow:
  - **Start Date (Inclusive):** This is the date on which the report should start if the custom time period option has been set.
  - **End Date (Inclusive):** This is the date on which the report should end if the custom time period option has been set.
- **Report time Period:** This is the time period for which the availability should be assessed. The default is **None**, meaning that the service or host's state at any time will be used in the calculations. You could, for example, set this to workhours to see the percentage of uptime during a time that the server is expected to be busy.
- **Assume Initial States:** If Nagios Core can't figure out the initial state of the service at the time the report begins until the first check, it will assume that it is based on the value of the **First Assumed Host State** or **First Assumed Service State** field.
- **Assume State Retention:** If Nagios Core was restarted one or more times during the reporting period, checking this option will make the report assume that the state before any restart was retained by Nagios Core until it started again; this is enabled with the `retain_state_information` directive in `nagios.cfg`.
- **Assume States During Program Downtime:** If Nagios Core finds that it was down for a period in its log files, it will assume the final state it reads from the host or service before it went down endured for that time.
- **Include Soft States:** Nagios Core will graph **SOFT** states, meaning that it will include state changes that do occur but return to their previous state before `max_check_attempts` is exhausted. Otherwise, it will only graph states that have endured right through the retry checks, or **HARD** states.
- **First Assumed Host State or First Assumed Service State:** This is the value Nagios Core should assume for the host or service state if it can't determine it from the log files.
- **Backtracked Archives:** This shows how many archived log files back the Nagios Core process should check to try and find initial states for the host or service.

## **There's more...**

You can choose to run the report for a hostgroup or servicegroup as well, which will yield an indexed table showing both the per-host or per-service percentage state time and also the average uptime for all the hosts or services in the group.

## See also

- The *Viewing and interpreting trends* recipe in this chapter
- The *Viewing and interpreting notification history* recipe in this chapter
- *Using authenticated contacts*, [Chapter 10](#), *Security and Performance*



# Viewing and interpreting trends

In this recipe, you'll learn how to use the **Host and Service State Trends** reporting tool on a host or service to show a graph of states over some fixed period of time. This can be useful to determine not only the overall availability, perhaps to meet the terms of a service-level agreement, but also to ascertain whether there are certain intervals or consistent times that the host enters a state that is not OK. It's a good way to look for patterns in the downtime of your hosts.

# Getting started

You will need access to the Nagios Core web interface and permission to run commands from the CGIs. The sample configuration installed by following the Quick Start Guide provides the nagiosadmin user all the necessary privileges when authenticated via HTTP.

If you find that you don't have this privilege, check the `authorized_for_all_services` and `authorized_for_all_hosts` directives in `/usr/local/nagios/etc/cgi.cfg` and include your username in both, for example, tom:

```
authorized_for_all_services=nagiosadmin,tom
authorized_for_all_hosts=nagiosadmin,tom
```

Alternatively, you should also be able to see a host or service's information if you are authenticating with the same username as the nominated contact for the host or service you want to check.

In this example, we'll view a month's history for the CPU load service on `roma.example.net`, a web server for which we've been running checks.

In Nagios 4.1.0, a new version of the **Trends** report was introduced using the new JSON data sources available to the CGIs. We'll demonstrate this one. The older version of the **Trends** report is still available. Just click on the **(Legacy)** link next to **Trends**.



# How to do it...

We can arrange a **Service State Trends** report for the last month for our roma.example.net server by following these steps:

1. Log in to the Nagios Core web interface.
2. Click on the **Trends** link in the left-hand side menu, beneath **Reports**:



3. A pop-up window should appear. Select the type of report, either **Host** or **Service**:

A screenshot of the 'Nagios Trends' pop-up window. The window has a title bar with 'Nagios Trends' and a close button. Inside, there are two tabs: 'Standard' (selected) and 'Advanced'. Below the tabs is a section titled 'URL for JSON CGIs' with a text input field containing 'http://olympus.example.net/nagios/cgi-bin/'. Below that is a section titled 'Report Type' with a dropdown menu. The dropdown menu is open, showing 'Service' as the selected option. A mouse cursor is pointing at the 'Service' option.

4. Select a specific host and/or service on which we need to report:

# Nagios Trends

Standard

Advanced

URL for JSON CGIs

http://olympus.example.net/nagios/cgi-bin/

Report Type

Service

Host

roma

Service

CPU load

5. Choose a time period for the report. In this example, we'll choose the **Last 7 Days** report, to see how the service has been performing over the past week. Click on **Apply** when you're done:

Nagios Trends ×

Standard

Advanced

URL for JSON CGIs

http://olympus.example.net/nagios/cgi-bin/

Report Type

Service

Host

roma

Service

CPU load

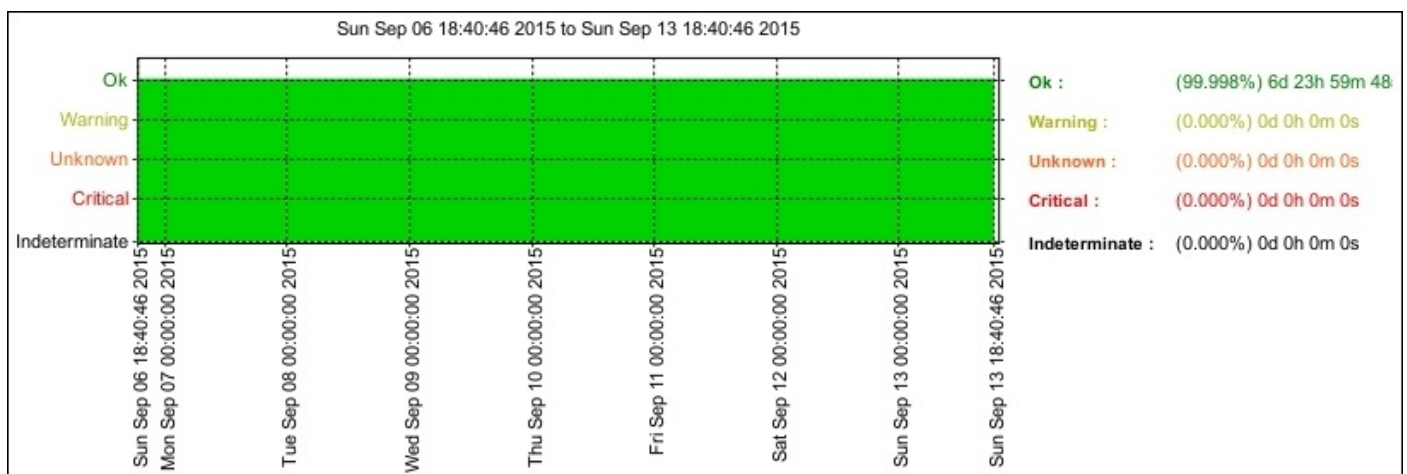
Report Period

Last 7 Days

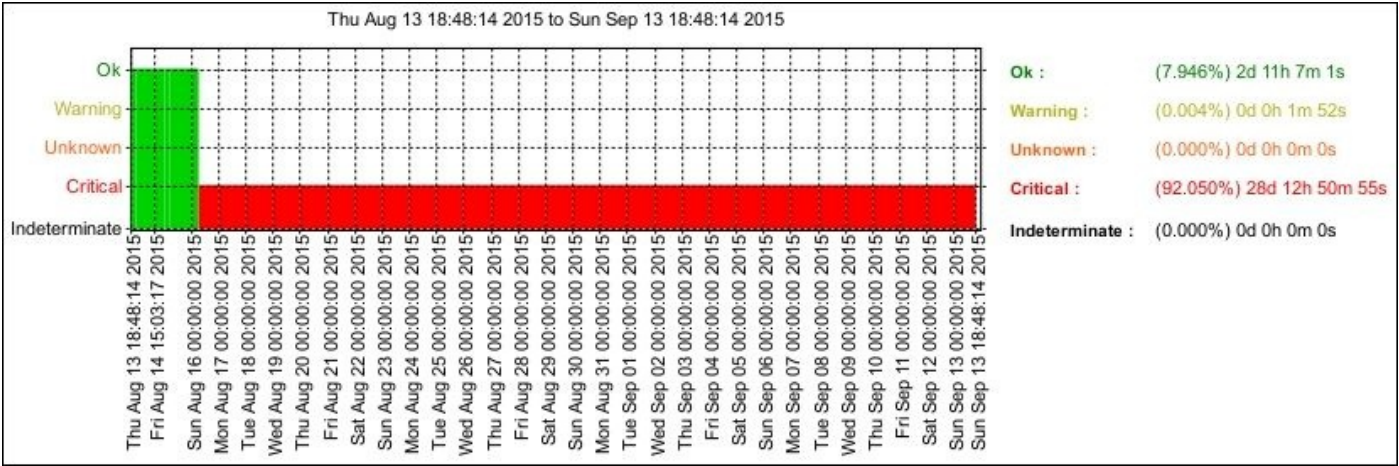
Apply

You should be presented with a graph showing the state of the host or service over time, along with the markings of time for when the state changed, and a percentage breakdown of the relative states to the right.

A healthy service might look like this, with a few blips or none at all:



A more problematic service might have long periods of time in the WARNING or CRITICAL state:



You can double-click on sections of the graph to *zoom in* on them, provided that you did not select the **Suppress image map** option on the **Advanced** page of the report dialog.

# How it works...

Nagios Core assembles state changes from its log files for the specified time period and constructs the graph of state changes by color, delineating the dates on the horizontal axis and at regular intervals. The trends graph, therefore, only works for times covered by your archived log files. The third step of building the report involves a lot of possible options, if you switch to the **Advanced** tab, which are as follows:

- **Report period:** This is the drop-down menu that allows us to choose a fixed period, for convenience, relative to the current date; alternatively, a custom time period may be used by selecting the final **CUSTOM REPORT PERIOD** option and selecting dates in the two fields that follow:
  - **Start Date (Inclusive):** This refers to the date on which the report should start, if the custom time period option has been set.
  - **End Date (Inclusive):** This is the date on which the report should end, if the custom time period option has been set.
  - **Assume Initial States:** If Nagios Core can't figure out the initial state of the service at the time the report begins until the first check, it will assume that it is based on the value of the **First Assumed State** field.
  - **Assume State Retention:** If Nagios Core was restarted one or more times during the reporting period, checking this option will make the report assume that the state before any restart was retained by Nagios Core until it started again; this is enabled with the `retain_state_information` directive in `nagios.cfg`.
- **Assume States During Program Downtime:** If Nagios Core finds it was down for a period in its log files, it will assume the final state it reads from the host or service while it was down for that time.
- **Include Soft States:** Nagios Core will graph `SOFT` states, meaning that it will include state changes that do occur but return to their previous state before `max_check_attempts` is exhausted. Otherwise, it will only graph states that have endured right through the retry checks, or `HARD` states.
- **First Assumed State:** This is the value Nagios Core should assume for the host or service state if it can't determine the value from the log files.
- **Backtracked Archives:** This shows the number of archived log files that the Nagios Core process should check to try and find the initial states for the host or service.
- **Suppress image map:** This prevents the graph from being clickable to zoom in on a particular region of it, perhaps for browser compatibility.
- **Suppress popups:** This prevents the graph from showing popups when hovering over its sections, perhaps for browser compatibility.

## There's more...

Note that it's important to ensure that checks have actually been running for the entire time for which you're running the report, as otherwise the **State Breakdowns** section will have distorted statistics. There is not much point running a yearly report for a host that has only existed for 6 months! In general, the more frequent and consistent your checks, the more accurate the trends graph will be.

## See also

- The *Viewing and interpreting availability reports* recipe in this chapter
- The *Viewing and interpreting notification history* recipe in this chapter





# Viewing and interpreting notification history

In this recipe, we'll see how to get both complete listings and convenient summaries of the alerts and notifications being generated by Nagios Core in response to hosts and services changing state. These options are all available under the **Reports** section of the sidebar:



It's important to distinguish between **alerts** and **notifications** in this section. An alert is generated in response to an event such as a host or service changing state. A notification, in turn, may or may not be generated as a response to that alert and can be sent to the appropriate contacts. The **SOFT** state changes constitute alerts; only **HARD** state changes generally generate notifications.

It's likely that a production monitoring server will not send notifications for every alert, particularly if you're making good use of the `max_check_attempts`, `scheduled_downtime`, and `problem_acknowledgement` features. So, you should make sure you're checking the correct section.

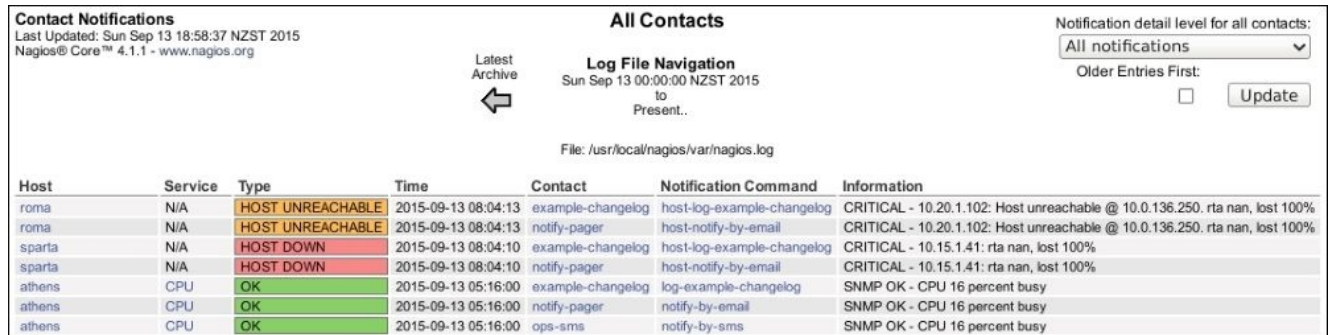
# Getting started

You will need access to the Nagios Core web interface and permission to run commands from the CGIs. The sample configuration installed by following the Quick Start Guide provides the nagiosadmin user all the necessary privileges when authenticated via HTTP.

# How to do it...

We can get an overview of the notifications, alerts, and other events being generated on the Nagios Core server as follows:

1. Start with the **Notifications** report. The resulting screen should show you the day's notifications as read from the current log file:



| Host   | Service | Type             | Time                | Contact           | Notification Command       | Information                                                                 |
|--------|---------|------------------|---------------------|-------------------|----------------------------|-----------------------------------------------------------------------------|
| roma   | N/A     | HOST UNREACHABLE | 2015-09-13 08:04:13 | example-changelog | host-log-example-changelog | CRITICAL - 10.20.1.102: Host unreachable @ 10.0.136.250. rta nan, lost 100% |
| roma   | N/A     | HOST UNREACHABLE | 2015-09-13 08:04:13 | notify-pager      | host-notify-by-email       | CRITICAL - 10.20.1.102: Host unreachable @ 10.0.136.250. rta nan, lost 100% |
| sparta | N/A     | HOST DOWN        | 2015-09-13 08:04:10 | example-changelog | host-log-example-changelog | CRITICAL - 10.15.1.41: rta nan, lost 100%                                   |
| sparta | N/A     | HOST DOWN        | 2015-09-13 08:04:10 | notify-pager      | host-notify-by-email       | CRITICAL - 10.15.1.41: rta nan, lost 100%                                   |
| athens | CPU     | OK               | 2015-09-13 05:16:00 | example-changelog | log-example-changelog      | SNMP OK - CPU 16 percent busy                                               |
| athens | CPU     | OK               | 2015-09-13 05:16:00 | notify-pager      | notify-by-email            | SNMP OK - CPU 16 percent busy                                               |
| athens | CPU     | OK               | 2015-09-13 05:16:00 | ops-sms           | notify-by-sms              | SNMP OK - CPU 16 percent busy                                               |

2. Note that if you're using the log rotation feature of Nagios Core, configured with the `log_rotation_method` directive, you can use the arrows above the table to navigate to the previous period's monitoring (generally, 24 hours' worth). Additionally, note that the information includes the following:

- Links to the hosts and services that generate them
- The contacts to which they were sent
- The notification commands that were run
- The output of the commands that prompted the alert and notification

Also note that you're able to filter for any particular kind of notification and change the sorting order using the form in the top-right corner:



3. Next, navigate to the **Alerts** section or the **Alerts | History** report. Rather than a tabular format, this shows you all the alerts generated by hosts and services as a list, showing red exclamation icons for CRITICAL or DOWN states, green icons for OK states, and others. The list is broken down by hour:

### Alert History

Last Updated: Sun Sep 13 19:05:41 NZST 2015  
Nagios® Core™ 4.1.1 - www.nagios.org

[View Status Detail For All Hosts](#)  
[View Notifications For All Hosts](#)

### All Hosts and Services

Latest Archive

**Log File Navigation**  
Sun Sep 13 00:00:00 NZST 2015  
to  
Present..

File: /usr/local/nagios/var/nagios.log

State type options:  
**All state types** ▼

History detail level for all hosts:  
**All alerts** ▼

☐ Hide Flapping Alerts  
☐ Hide Downtime Alerts  
☐ Hide Process Messages  
☐ Older Entries First

**Update**

---

**September 13, 2015 19:00**

!

[2015-09-13 19:05:30] SERVICE ALERT: sparta;PING;WARNING;SOFT;1;WARNING - 10.19.2.84: rta 21.776ms, lost 20%

!

[2015-09-13 19:05:30] SERVICE ALERT: athens;PING;WARNING;SOFT;1;WARNING - 10.16.5.24: rta 16.021ms, lost 20%

!

[2015-09-13 19:05:29] SERVICE ALERT: roma;PING;WARNING;SOFT;1;WARNING - 10.16.6.2: rta 23.002ms, lost 20%

[2015-09-13 19:05:22] SERVICE ALERT: sparta;PING;OK;SOFT;3;OK - 10.19.2.44: rta 33.859ms, lost 0%

[2015-09-13 19:05:22] SERVICE ALERT: roma;PING;OK;SOFT;2;OK - 10.16.5.3: rta 21.624ms, lost 0%

[2015-09-13 19:05:22] HOST ALERT: athens;UP;SOFT;2;OK - 10.47.0.8: rta 30.161ms, lost 0%

---

**September 13, 2015 18:00**

[2015-09-13 18:59:52] HOST ALERT: sparta;UP;SOFT;2;OK - 10.19.2.1: rta 12.182ms, lost 0%

[2015-09-13 18:59:43] SERVICE ALERT: sparta;PING;OK;SOFT;2;OK - 10.19.2.77: rta 43.937ms, lost 0%

[2015-09-13 18:59:37] SERVICE ALERT: roma;PING;OK;SOFT;2;OK - 10.19.2.71: rta 24.586ms, lost 0%

!

[2015-09-13 18:59:33] HOST ALERT: roma;DOWN;SOFT;1;CRITICAL - 10.34.0.52: rta nan, lost 100%

Note that you're again able to filter for specific alert types with the form in the top-right corner.

- The other two items in the **Alerts** menu allow us to generate complex reports with many criteria. The first, **Summary**, presents its results in a tabular format according to the criteria entered in a form. For now, try clicking on the **Create Summary Report** at the bottom of the first page, with all the fields on their defaults, to get a feel of the results it generates:

### Alert Summary Report

Last Updated: Sun Sep 13 19:17:16 NZST 2015  
Nagios® Core™ 4.1.1 - www.nagios.org

### Most Recent Alerts

2015-09-06 19:17:15 to 2015-09-13 19:17:15  
Duration: 7d 0h 0m 0s

### Report Options Summary:

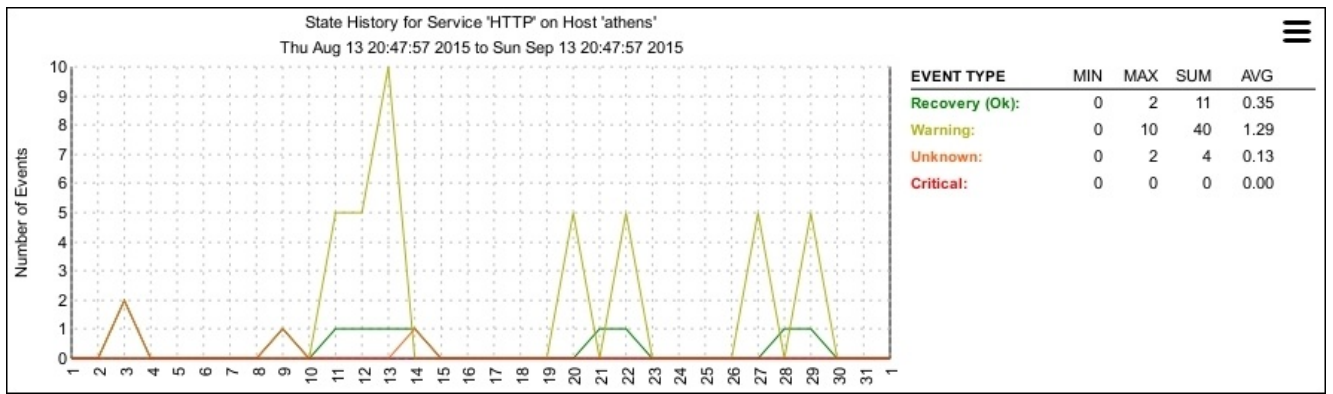
**Alert Types:** Host & Service Alerts  
**State Types:** Soft & Hard States  
**Host States:** Up, Down, Unreachable  
**Service States:** Ok, Warning, Unknown, Critical

**Generate New Report**

Displaying most recent 25 of 57576 total matching alerts

| Time                | Alert Type    | Host    | Service        | State    | State Type | Information                                               |
|---------------------|---------------|---------|----------------|----------|------------|-----------------------------------------------------------|
| 2015-09-13 19:17:07 | Service Alert | athens  | PING           | OK       | SOFT       | OK - 192.0.2.73: rta 22.707ms, lost 0%                    |
| 2015-09-13 19:16:59 | Service Alert | olympus | PING           | WARNING  | SOFT       | WARNING - 192.0.2.2: rta 7.329ms, lost 20%                |
| 2015-09-13 19:16:58 | Service Alert | roma    | EXAMPLE_SCRIPT | UNKNOWN  | SOFT       | EXAMPLE_SCRIPT UNKNOWN - Couldn't get prompt after output |
| 2015-09-13 19:16:53 | Service Alert | athens  | PING           | OK       | SOFT       | OK - 192.0.2.80: rta 42.954ms, lost 0%                    |
| 2015-09-13 19:16:39 | Host Alert    | mycenae | N/A            | DOWN     | SOFT       | CRITICAL - 192.0.2.40: rta nan, lost 100%                 |
| 2015-09-13 19:16:29 | Service Alert | mycenae | PING           | CRITICAL | SOFT       | CRITICAL - 192.0.2.40: rta nan, lost 100%                 |

The **Histogram** report also generates a graph similar to the following one, showing a breakdown of the alerts generated for nominated hosts or services over a fixed period of time:



## How it works...

Nagios Core saves the data for both alerts and notifications on a long-term basis, allowing reports like this to be generated dynamically from its archives. Note that the data presented in these reports is for alerts and notifications, not the actual states of the services over time. To get useful statistics such as the percentage of uptime, you'll want to generate an **Availability** report, which is discussed in the *Viewing and interpreting availability reports* recipe in this chapter.

The same notifications data can also be translated into a MySQL database format via the NDOUtils extension so that reading the alerts and notifications data programmatically to generate custom reports is possible. Refer to the *Reading status into a MySQL database with NDOUtils* recipe in [Chapter 11](#), *Automating and Extending Nagios Core* for more information on how to do this.

## There's more...

The remaining item in the menu is the **Event Log** item, which presents a more comprehensive summary of Nagios Core's overall activity without filtering only for alerts or notifications. This screen can include information such as the application of external commands. It tends to be quite verbose, but it's a useful way to read the `nagios.log` file from the web interface. Your username will need to be included in the `authorized_for_system_information` directive in `/usr/local/nagios/etc/cgi.cfg` to use this:

```
authorized_for_system_information=nagiosadmin,tom
```

## See also

- The *Viewing and interpreting availability reports* recipe in this chapter
- The *Viewing and interpreting trends* recipe in this chapter
- *Reading status into a MySQL database with NDOUtils*, [Chapter 11](#), *Automating and Extending Nagios Core*





# Adding comments on hosts or services in the web interface

In this recipe, you'll learn how to add comments to hosts or services in the Nagios Core web interface to keep track of information about them for all the web interface's users.

# Getting started

You will need access to the Nagios Core web interface and permission to run commands from the CGIs. The sample configuration installed by following the Quick Start Guide provides the nagiosadmin user all the necessary privileges when authenticated via HTTP. You will also need at least one host or service.

If you find that you don't have this privilege, check the `authorized_for_all_service_commands` and `authorized_for_all_host_commands` directives in `/usr/local/nagios/etc/cgi.cfg` and include your username in both, for example, tom:

```
authorized_for_all_service_commands=nagiosadmin,tom
authorized_for_all_host_commands=nagiosadmin,tom
```

# How to do it...

We can add a comment to a host or service by following these steps:

1. Log in to the Nagios Core web interface, and click on the hostname or service description on which you wish to leave a comment. You can get to this via the **Hosts** or **Services** menu items. Here, I'm leaving a comment on my `sparta.example.net` host:

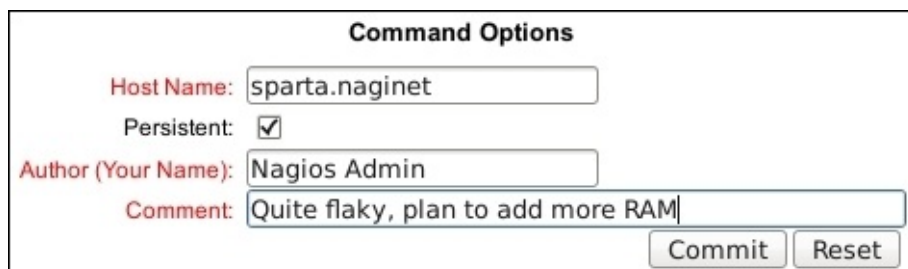


2. Click on the **Add a new comment** link at the bottom of the page:



3. Fill out the resulting form. Include the following details:
  - **Host Name:** This refers to the hostname for the host or service on which the comment should be made. This should be filled in automatically.
  - **Persistent:** Check this box if you would like the comment to remain on the host even after Nagios Core is restarted.
  - **Author (Your Name):** This is the name of the person acknowledging the fault. This should default to your name or username; it may be grayed out and unchangeable, depending on the value of `lock_author_names` in `/usr/local/nagios/etc/cgi.cfg`.
  - **Comment:** This is the text of the comment itself.

Note that explanatory notes also appear on the right-hand side of the command description. Click on **Commit** when you're done:

A screenshot of the 'Command Options' form. It contains the following fields:

- Host Name:** A text box containing 'sparta.naginet'.
- Persistent:** A checkbox that is checked.
- Author (Your Name):** A text box containing 'Nagios Admin'.
- Comment:** A text box containing 'Quite flaky, plan to add more RAM'.

At the bottom right of the form are two buttons: 'Commit' and 'Reset'.

With this done, a comment should be added to the host or service. It may take a little while for the command to be processed. You will find that there is an acknowledgment icon

added to the host or service's link in its menu, and that a comment has been added:



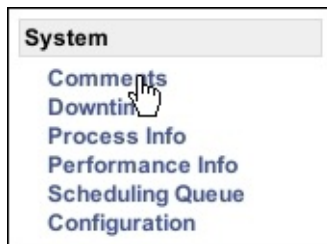
## How it works...

Like most changes issued from the Nagios Core web interface, adding a comment to a host or service is a command, issued by the server for processing along with its primary task of executing plugins and recording states. This is why it can sometimes take a few seconds to apply even on an idle server. This is all written to the command file, by default stored at `/usr/local/nagios/var/rw/nagios.cmd`, which the Nagios Core system regularly reads. This is a **named pipe** and cannot normally be read from the command line.

When the command is executed, a comment is added to the host or service that can be viewed by anyone with appropriate permissions in the web interface.

# There's more...

You can view a list of all comments with the **Comments** link in the sidebar:



This will bring up a complete list of comments for all hosts and services, including both automated and manual ones:

[\[ Host Comments | Service Comments \]](#)

### Host Comments

 [Add a new host comment](#)

| Host Name              | Entry Time          | Author | Comment                           | Comment ID | Persistent | Type | Expires | Actions                                                                             |
|------------------------|---------------------|--------|-----------------------------------|------------|------------|------|---------|-------------------------------------------------------------------------------------|
| <a href="#">sparta</a> | 2015-09-13 10:20:42 | tom    | Quite flaky, plan to add more RAM | 172        | Yes        | User | N/A     |  |

### Service Comments

 [Add a new service comment](#)

| Host Name              | Service              | Entry Time          | Author | Comment                    | Comment ID | Persistent | Type            | Expires | Actions                                                                               |
|------------------------|----------------------|---------------------|--------|----------------------------|------------|------------|-----------------|---------|---------------------------------------------------------------------------------------|
| <a href="#">athens</a> | <a href="#">HTTP</a> | 2015-09-04 16:38:50 | tom    | Pending fix from developer | 161        | Yes        | Acknowledgement | N/A     |  |

## See also

- The *Acknowledging a problem via the web interface* recipe in this chapter





# Viewing configuration in the web interface

In this recipe, you'll learn how to view a table of all the objects currently configured for the running Nagios Core instance. This is a very convenient way to view how the system has understood your configuration, which really helps make sense of things if you use a lot of tricks such as object inheritance and patterns for hostnames.

Because it's tucked away at the bottom of the Nagios Core web interface's navigation menu, some administrators aren't aware of this feature. It's very simple to use as there are only really two steps involved.

# Getting started

You will need access to the Nagios Core web interface and permission to run commands from the CGIs. The sample configuration installed by following the Quick Start Guide provides the nagiosadmin user all the necessary privileges when authenticated via HTTP.

If you find that you don't have this privilege, check the `authorized_for_configuration_information` directive in `/usr/local/nagios/etc/cgi.cfg` and include your username in it, for example, tom:

```
authorized_for_configuration_information=nagiosadmin,tom
```

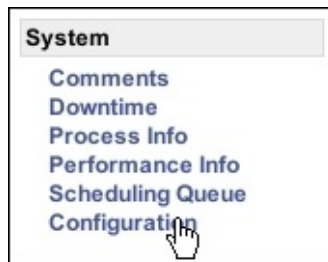
## Note

Changes to this file will require us to restart the Nagios Core server.

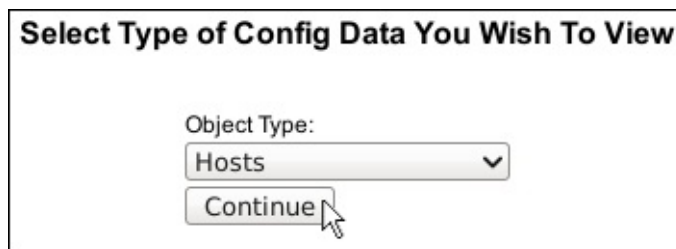
# How to do it...

We can view the configuration of different object types in the web interface as follows:

1. Log in to the Nagios Core web interface, and click on the **Configuration** item in the navigation menu:



2. Select the object type you wish to view, and click on **Continue**. In this example, I want to view all the hosts I have defined:



You should be presented with a complete table showing the directive values of all the hosts defined in your system. It will likely be wider than the screen, and will hence require horizontal scrolling:

| Hosts          |                   |            |                   |                              |              |                     |                |                |                    |              |          |
|----------------|-------------------|------------|-------------------|------------------------------|--------------|---------------------|----------------|----------------|--------------------|--------------|----------|
| Host Name      | Alias/Description | Address    | Importance (Host) | Importance (Host + Services) | Parent Hosts | Max. Check Attempts | Check Interval | Retry Interval | Host Check Command | Check Period | Observed |
| localhost-orig | localhost-orig    | 127.0.0.1  | 0                 | 0                            |              | 10                  | 0h 5m 0s       | 0h 1m 0s       | check-host-alive   | 24x7         | Yes      |
| roma           | roma              | 192.0.2.61 | 0                 | 0                            |              | 1                   | 0h 1m 0s       | 0h 1m 0s       | check-mk-host-ping | 24X7         | Yes      |

# How it works...

This is simply a convenient shortcut to view the configuration of your hosts as Nagios Core understands them. The following object types can be viewed:

- Hosts
- Host Dependencies
- Host Escalations
- Host Groups
- Services
- Service Groups
- Service Dependencies
- Service Escalations
- Contacts
- Contact Groups
- Time Periods
- Commands
- Command Expansion

This shortcut is particularly convenient for definitions that can sometimes be quite complex, such as hosts and services that inherit from templates or time period definitions. It's a good way to verify that Nagios Core has interpreted your configuration in the way you expected.

## There's more...

It's important to note that the output doesn't actually tell you anything about the state of hosts or services; it only shows how they're configured. If you need to get both configuration and state information in an accessible format by a custom script for processing, an excellent way to arrange this is with the NDOUtils add-on, which allows you to translate this information into a MySQL database format. Refer to the *Reading status into a MySQL database with NDOUtils* recipe in [Chapter 11](#), *Automating and Extending Nagios Core* for more information on how to do this.

## See also

- *Reading a status into a MySQL database with NDOUtils, [Chapter 11](#), Automating and Extending Nagios Core*
- *Writing customized Nagios Core reports, [Chapter 11](#), Automating and Extending Nagios Core*





# Scheduling checks from the web interface

In this recipe, you'll learn how to manually schedule checks of hosts and services from the web interface, overriding the automatic scheduling normally done by Nagios Core. This can be convenient to hurry along checks for hosts or services that have just been added, or which have just had problems, or to force a check to be made even when active checks are otherwise disabled on a host for whatever reason.

In this example, we'll schedule a check for a service, but checks for hosts can be established in just the same way.

# Getting started

You will need access to the Nagios Core web interface and permission to run commands from the CGIs. The sample configuration installed by following the Quick Start Guide provides the nagiosadmin user all the necessary privileges when authenticated via HTTP.

If you find that you don't have this privilege, check the `authorized_for_all_service_commands` and `authorized_for_all_host_commands` directives in `/usr/local/nagios/etc/cgi.cfg` and include your username in both, for example, tom:

```
authorized_for_all_service_commands=nagiosadmin,tom
authorized_for_all_host_commands=nagiosadmin,tom
```

## Note

Changes to this file will require us to restart the Nagios Core server.

# How to do it...

We can schedule a check for an existing service by following these steps:

1. Log in to the Nagios Core web interface, and find the link to the host or service for which you want to schedule a check. In this example, we're forcing a check of the LOAD service on the sparta.example.net host:

|        |        |   |    |
|--------|--------|---|----|
| sparta | BACKUP | ? | OK |
|        | DISK   |   | OK |
|        | HTTP   | ? | OK |
|        | LOAD   | ? | OK |

2. Click on the **Re-schedule the next check of this service** (or **host**) in the right-hand side menu:

| Service Commands |                                                            |
|------------------|------------------------------------------------------------|
| ✗                | Disable active checks of this service                      |
| 🕒                | <a href="#">Re-schedule the next check of this service</a> |
| ?                | Submit passive check result for this service               |
| ✗                | Stop accepting passive checks for this service             |
| ✗                | Stop obsessing over this service                           |
| ✗                | Disable notifications for this service                     |
| 📧                | Send custom service notification                           |
| 🕒                | Schedule downtime for this service                         |
| ✗                | Disable event handler for this service                     |
| ✗                | Disable flap detection for this service                    |

3. Complete the resulting form. Include the following details:
  - **Host Name:** This is the hostname for the host or service that has the problem. This should be filled in automatically.
  - **Service:** This refers to the service description for the service having problems, if applicable. This will also be filled in automatically.
  - **Check Time:** This is the time at which Nagios Core should reschedule the check.
  - **Force Check:** Whether to force a check, regardless of active checks being disabled for this host or service.

Note that explanatory notes also appear on the right-hand side of the command description. Click on **Commit** when you're done:

**Command Options**

Host Name:

sparta

Service:

LOAD

Check Time:

2015-09-13 21:23:05

Force Check:

☒

Commit

Reset

With this done, after Nagios Core processes the new command, the service should now show the new time for its next scheduled check. The check will be run at the nominated time, and its return values will be processed normally.

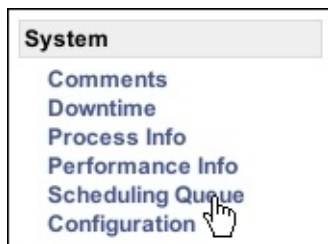
## How it works...

Scheduling checks in this manner is a means of *jumping the queue* for a particular host or service, generally an automated process that Nagios Core figures out based on host and service directives such as `check_interval` and `retry_interval`.

Like all external commands issued from the web interface, a command is written to the commands file, usually at `/usr/local/nagios/var/rw/nagios.cmd`, by the system for processing.

# There's more

We can inspect the enqueued checks by clicking on the **Scheduling Queue** link under the **System** section of the sidebar:



This will bring up a list of the pending checks and the time at which they'll take place:

| Entries sorted by next check time (ascending) |                |                     |                     |        |               |         |
|-----------------------------------------------|----------------|---------------------|---------------------|--------|---------------|---------|
| Host ↕                                        | Service ↕      | Last Check ↕        | Next Check ↕        | Type   | Active Checks | Actions |
| sparta                                        | LOAD           | 2015-09-13 21:22:10 | 2015-09-13 21:26:35 | Normal | ENABLED       | ✗ ⌚     |
| athens                                        |                | 2015-09-13 21:21:36 | 2015-09-13 21:26:36 | Normal | ENABLED       | ✗ ⌚     |
| roma                                          | EXAMPLE_SCRIPT | 2015-09-13 21:21:36 | 2015-09-13 21:26:36 | Normal | ENABLED       | ✗ ⌚     |
| sparta                                        |                | 2015-09-13 21:21:37 | 2015-09-13 21:26:37 | Normal | ENABLED       | ✗ ⌚     |

We can also get more detailed information about the queuing process and performance using the `-s` flag with the `nagios` command:

```
# /usr/local/nagios/bin/nagios -s /usr/local/nagios/etc/nagios.cfg
```

This will print a lot of information to the terminal, too much to reproduce completely here! It is probably helpful to pipe it through a paging tool such as `less` to read it:

```
# /usr/local/nagios/bin/nagios -s /usr/local/nagios/etc/nagios.cfg | less
Nagios Core 4.1.1
Copyright (c) 2009-present Nagios Core Development Team and Community
Contributors
Copyright (c) 1999-2009 Ethan Galstad
Last Modified: 08-19-2015
License: GPL
```

Website: <https://www.nagios.org>

Timing information on object configuration processing is listed below. You can use this information to see if precaching your object configuration would be useful.

Object Config Source: Config files (uncached)

OBJECT CONFIG PROCESSING TIMES (\* = Potential for precache savings with `-u` option)

```
-----
Read:                0.109772 sec
```

|                       |          |     |   |
|-----------------------|----------|-----|---|
| Resolve:              | 0.001497 | sec | * |
| Recomb Contactgroups: | 0.000629 | sec | * |
| Recomb Hostgroups:    | 0.046730 | sec | * |
| Dup Services:         | 0.006301 | sec | * |
| ...                   |          |     |   |

## See also

- The *Viewing and interpreting notification history* recipe in this chapter
- *Checking Nagios performance with Nagiosstats*, [Chapter 11](#), *Automating and Extending Nagios Core*
- *Allowing passive checks*, [Chapter 11](#), *Automating and Extending Nagios Core*
- *Running passive checks from a remote host with NSCA*, [Chapter 11](#), *Automating and Extending Nagios Core*





# Acknowledging a problem via the web interface

In this recipe, you'll learn how to acknowledge problems with a host or service, in order to prevent further notifications coming from it and to signal that the problem is being worked on. This is useful if more than one administrator has access to the Nagios Core web interface. This is done in order to prevent more than one administrator trying to fix a problem and to prevent unnecessary notifications for a longer term problem after the operations team has been made aware of it. The exact changes in behavior that the acknowledgement causes are defined at the time it's submitted.

# Getting started

To acknowledge a notification, there needs to be at least one host or service suffering problems. Any host or service in the WARNING, CRITICAL, or UNKNOWN state can be acknowledged.

You will need access to the Nagios Core web interface and permission to run commands from the CGIs. The sample configuration provides the nagiosadmin user all the necessary privileges when authenticated via HTTP.

If you find that you don't have this privilege, check the `authorized_for_all_service_commands` and `authorized_for_all_host_commands` directives in `/usr/local/nagios/etc/cgi.cfg` and include your username in both, for example, tom:

```
authorized_for_all_service_commands=nagiosadmin,tom
authorized_for_all_host_commands=nagiosadmin,tom
```

## Note

Changes to this file will require you to restart the Nagios Core server.

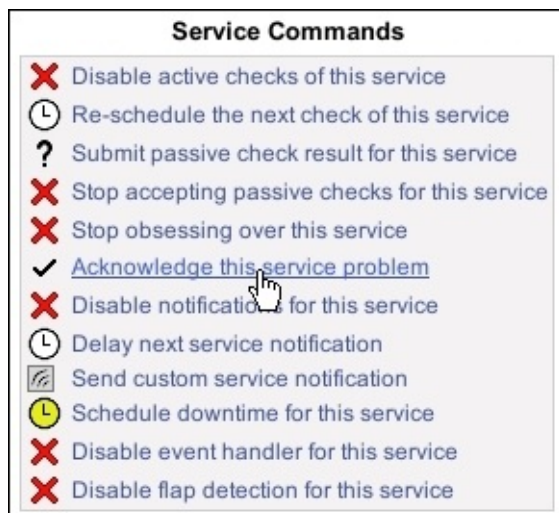
# How to do it...

We can acknowledge a host or service problem as follows:

1. Log in to the Nagios Core web interface, and click on the **Hosts** or **Services** section in the left-hand side menu, to visit the host or service having problems. In this example, I'm acknowledging a problem with the APT service on my sparta machine; Nagios is reporting upgrades that are available, but they're not critical and I can't run them right now, so I'm going to acknowledge them so that other administrators know I've got a plan for the problem:

|        |     |         |                     |               |     |
|--------|-----|---------|---------------------|---------------|-----|
| sparta | APT | WARNING | 2015-09-13 21:30:54 | 0d 11h 2m 22s | 5/5 |
|--------|-----|---------|---------------------|---------------|-----|

2. Click on the **Acknowledge this service problem** link under the **Service Commands** menu:

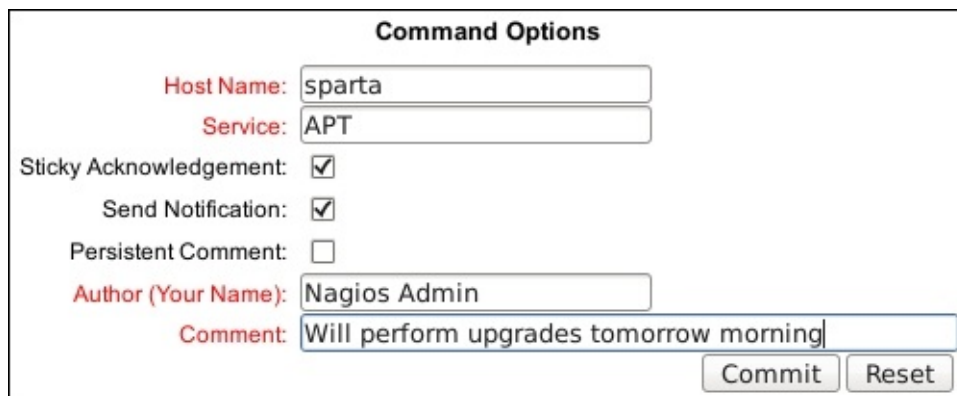


3. Complete the resulting form. Include the following details:
  - **Host Name:** This refers to the hostname for the host or service that has the problem. This should be filled in automatically.
  - **Service:** This refers to the service description for the service having problems, if applicable. This will also be filled in automatically.
  - **Sticky Acknowledgement:** If this is checked, notifications will be suppressed until the problem is resolved; the acknowledgement will not go away if Nagios Core is restarted.
  - **Send Notification:** If this is checked, an **ACKNOWLEDGEMENT** notification will be sent to all the contacts and contact groups for the host or service.
  - **Persistent Comment:** Acknowledgements always leave comments on the host or service in question. By default, these are removed when the host or service

recovers, but if you wish you can arrange to have them take the form of a permanent comment.

- **Author (Your Name):** This is the name of the person acknowledging the fault. This should default to your name or username; it may be grayed out and unchangeable, depending on the value of `lock_author_names` in `/usr/local/nagios/etc/cgi.cfg`.
- **Comment:** This is an explanation of the acknowledgement; this is generally a good place to put expected times to resolution and what's being done about the problem.

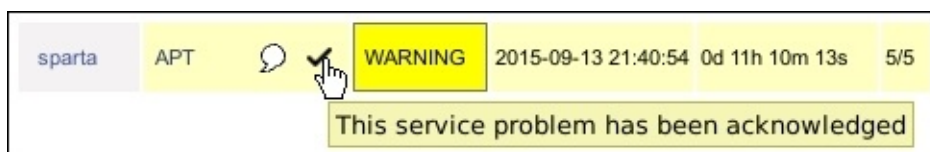
Note that explanatory notes also appear on the right-hand side of the command description. Click on **Commit** when you're done:



The screenshot shows the 'Command Options' form in Nagios. It contains the following fields and controls:

- Host Name:**
- Service:**
- Sticky Acknowledgement:** ☒
- Send Notification:** ☒
- Persistent Comment:** ☐
- Author (Your Name):**
- Comment:**
- Buttons:**

With this done, the effects you chose for the downtime should take effect shortly. It may take a little while for the command to be processed. You will find that there is an acknowledgment icon added to the host or service's link in its menu, and that a comment has been added:



Your contacts may also receive an ACKNOWLEDGEMENT notification:

Date: Mon, 1 Jun 2015 18:19:59 +1200 (NZST)  
From: nagios@olympus.example.net  
To: nagios@example.net  
Subject: \*\* ACKNOWLEDGEMENT Service Alert: sparta/APT is WARNING \*\*

\*\*\*\*\* Nagios \*\*\*\*\*

Notification Type: ACKNOWLEDGEMENT

Service: APT  
Host: sparta  
Address: 192.0.2.21  
State: WARNING

Date/Time: Sun Sept 13 21:41:57 NZST 2015

Additional Info:

APT WARNING: 34 packages available for upgrade (0 critical updates).

## How it works...

Like most changes issued from the Nagios Core web interface, acknowledging a host or service is a command issued by the server for processing, along with its primary task of executing plugins and recording states. This is why it can sometimes take a few seconds to apply even on an idle server. This is all written to the command file, by default stored at `/usr/local/nagios/var/rw/nagios.cmd`, which the Nagios Core system regularly reads.

When the command is executed, and the **Sticky Acknowledgement** option is selected, the effect is that the filters normally applied by Nagios Core before dispatching a notification will notice that the host or service has had its problems acknowledged and will therefore prevent the notification from being sent.

When the host recovers (a RECOVERY event), a notification to that effect will be sent, and the **Acknowledgement** will be removed, as it's no longer needed. Future problems with the host or service, even if they're exactly the same problem, will need to be acknowledged.

## There's more...

Acknowledging host and service problems is a really good habit to get into for administrators, particularly if they are working in a team, as it may prevent a lot of confusion or duplication of work. It's therefore helpful in the same way comments are, for keeping your team informed about what's going on with the network.

The distinction between scheduled downtime and acknowledgements is important. Downtime is intended to be for planned outages to a host or service; acknowledgement is intended as a means of notification that an unanticipated problem has occurred and is being worked on. For this reason, you should avoid getting in the habit of using the scheduled downtime feature to suppress notifications for a problem that you didn't expect; this is what acknowledgements are for.

If we're getting nagged repeatedly by notifications for problems that we already know about, as well as using acknowledgements, it may be appropriate to review the `notification_interval` directive for the applicable hosts or services in order to limit the frequency at which notifications are sent. For example, to only send repeat notifications every 4 hours, we could write the following code:

```
define host {  
    ...  
    notification_interval 240  
}
```

For low-priority hosts, for example, it's perhaps not necessary to send notifications every 10 minutes!

Do not confuse this directive with `notification_period`, which is used to define the times during which notifications may be sent, another directive that may require review:

```
define host {  
    ...  
    notification_period 24x7  
}
```



## See also

- *Scheduling downtime for a host or service*, [Chapter 3](#), *Working with Checks and States*
- The *Adding comments on hosts or services in the web interface* recipe in this chapter



# Chapter 8. Managing Network Layout

In this chapter, we will cover:

- Creating a network host hierarchy
- Using the network map
- Choosing icons for hosts
- Establishing a host dependency
- Establishing a service dependency
- Monitoring individual nodes in a cluster
- Using the network map as an overlay

# Introduction

While Nagios Core is still very useful when configured to monitor only a simple list of hosts and services, it includes some optional directives that allow you to define some structural and functional properties of the monitored network—specifically, how hosts and services interrelate. Describing this structure in the configuration calls for some additional intelligent behavior in the monitoring and notification that Nagios Core performs.

There are two main approaches to working with network structure in Nagios Core:

- Host **parent** definitions allow an administrator to define the hierarchy of connectivity to monitored hosts from the “point of view” of the Nagios Core server. An example might be a server with the monitored address in another subnet linked to the Nagios Core server by a router. If the router enters the DOWN state, this triggers Nagios Core’s **host reachability logic** to automatically determine which hosts will become inaccessible and flags these as UNREACHABLE rather than DOWN, allowing refined notification behavior.
- Host and service **dependencies** allow us to formalize relationships between hosts or services, usually for the purposes of suppressing unnecessary notifications. An example might be a service that tests a login to a mail service that itself requires a database service to work properly. If Nagios Core finds that the database and login services are both down, a service dependency allows you to suppress the notification about the login service; the administrator would, therefore, only be notified about the database service being down, which is more likely to be the actual problem.

There is some overlap of functionality here, but the general pattern is that host parent definitions describe the structure of your network from the vantage point of your monitoring server and host and service dependencies describe the way it functions independent of the monitoring server. We will define both parent definitions and dependencies in this chapter, with the primary goal of filtering and improving the notifications that Nagios Core sends in response to failed checks, which can assist greatly in diagnosing problems.

We’ll also look at another, more subtle benefit of establishing host parent definitions in making the **network map** of the Nagios Core web interface useful and, once a basic hierarchy is set up, we’ll show how to customize the map’s appearance (including defining icons for hosts) to make it generally useful as a network **weather map**.



# Creating a network host hierarchy

In this recipe, you'll learn how to establish a parent-child relationship for two hosts in a very simple network in order to take advantage of Nagios Core's reachability logic. Changing this configuration is very simple; it involves adding only one directive and optionally changing some notification options.

# Getting ready

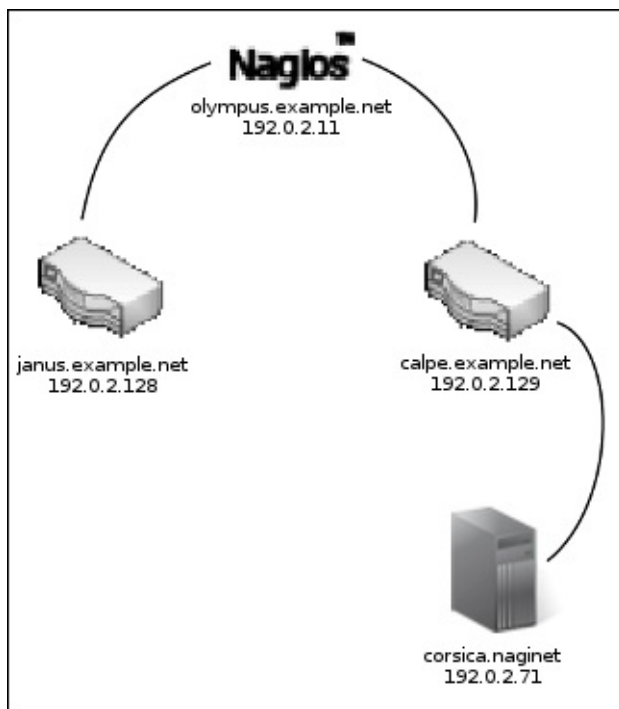
You will need to run Nagios Core 4.0 or a newer server and have at least two hosts, one of which is only reachable via the other. The host that allows communication with the other is the **parent host**. You should be reasonably confident that a loss of connectivity to the parent host necessarily implies that the child host will become unreachable from the monitoring server.

Access to the web interface of Nagios Core would also be useful as making this change will change the appearance of the network map; this is discussed in the “Using the network map” recipe in this chapter.

Our example will use a Nagios Core monitoring server, `olympus.example.net`, which will monitor the following three hosts:

- `calpe.example.net`, which is a router
- `janus.example.net`, which is another router
- `corsica.example.net`, which is a web server

The hosts are connected as in the following diagram:



Note that the Nagios Core server `olympus.example.net` is only able to communicate with the `corsica.example.net` webserver if the `calpe.example.net` router works correctly. If `calpe.example.net` were to enter a DOWN state, we would see `corsica.example.net` enter a DOWN state too. Take a look at the following screenshot:

| Host  |  | Status  | Last Check  |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| calpe.example.net                                                                      |  | DOWN                                                                                     | 10-05-2015 14:11:00                                                                          |
| corsica.example.net                                                                    |  | DOWN                                                                                     | 10-05-2015 14:12:36                                                                          |
| janus.example.net                                                                      |  | UP                                                                                       | 10-05-2015 14:12:41                                                                          |

This is a little misleading as we don't actually know whether `corsica.example.net` is down. It might be, but with the router between the hosts not working correctly, Nagios Core has no way of knowing. A more informative and accurate status for the host would be `UNREACHABLE`; this is what the configuration we're about to add will arrange.



# How to do it...

We can configure a parent-child relationship for our two hosts like so:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to its directory instead and run the following line of code:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the child host. In our example, the child host is `corsica.example.net`, which is a web server. The host definition might look something similar to this:

```
define host {
    use          linux-server
    host_name    corsica.example.net
    alias        corsica
    address      192.0.2.71
}
```

3. Add a new `parents` directive to the host's definition and give it the `host_name` value of the host on which it is dependent for connectivity. In our example, this host is `calpe.example.net`. Execute the following:

```
define host {
    use          linux-server
    host_name    corsica.example.net
    alias        corsica
    address      192.0.2.71
    parents      calpe.example.net
}
```

4. Validate the configuration and restart the Nagios Core server, as follows:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, if the parent host enters a `DOWN` state and the child host can't be contacted, then the child host will enter an `UNREACHABLE` state rather than also being flagged as `DOWN`:

| Host ↕              | Status ↕                                                                                        | Last Check ↕        |
|---------------------|-------------------------------------------------------------------------------------------------|---------------------|
| calpe.example.net   |  DOWN        | 10-05-2015 14:15:05 |
| corsica.example.net |  UNREACHABLE | 10-05-2015 14:15:05 |
| janus.example.net   |  UP          | 10-05-2015 14:15:05 |

The child host's contacts will also receive `UNREACHABLE` notifications instead of `DOWN` notifications for the child host, provided the `u` flag is included in `notification_options` for the host and `host_notification_options` for the contacts. Refer to the *Specifying which states to be notified about* recipe in [Chapter 4, Configuring Notifications](#), for details

on this.

# How it works...

This is a simple application of Nagios Core's **reachability logic**. When the check to `calpe.example.net` fails for the first time, Nagios Core notes that it is a **parent host** for one **child host**: `corsica.example.net`. If during checks for the child host it finds that it cannot communicate with it, then instead of the `DOWN` state, it flags an `UNREACHABLE` state, firing a different notification event.

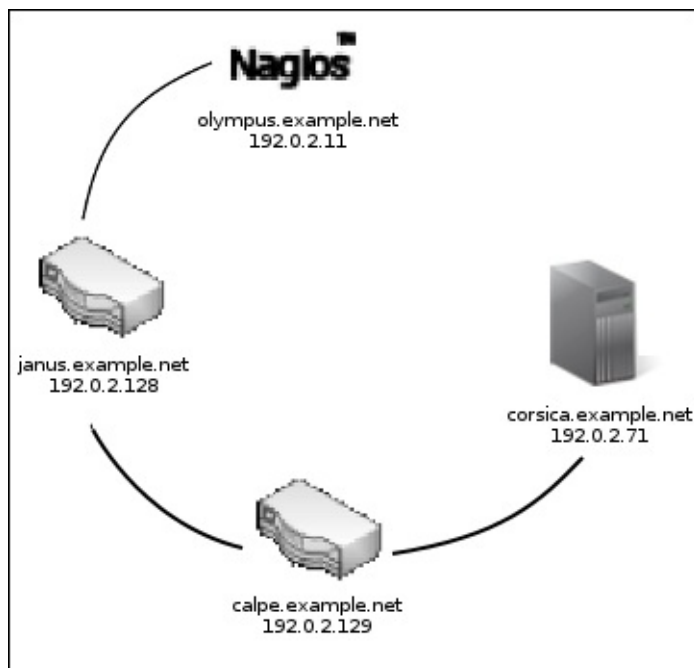
The primary advantages to this are two fold:

- The `DOWN` notification is only sent for the nearest problem parent host. All other hosts beyond this host fire `UNREACHABLE` notifications. This means that Nagios Core's reachability logic automatically determines the point of failure from its perspective, which can be very handy in diagnosing which host is actually experiencing a problem.
- If the host is parent to a large number of other hosts, the configuration can be arranged not to send urgent notifications for `UNREACHABLE` hosts. There may not be much point in sending a hundred pager or e-mail messages to an administrator when a very central router goes down; they know there are problems with the downstream hosts, so all we would be doing is distracting them with useless information.

With a little planning and some knowledge of the network, all we need to do is add a few parents directives to host definitions to build a simple network structure and Nagios Core behaves much more intelligently as a result. This is one of the easiest ways to refine the notification behavior of Nagios Core; it can't be recommended enough!

## There's more...

Note that a child host can itself be a parent to other hosts in turn, allowing a nesting network structure. Perhaps in another situation, we might find the `corsica.example.net` server is two routers away from the monitoring server, as shown in the following diagram:



In this case, not only is `corsica.example.net` the child host of `calpe.example.net`, but `calpe.example.net` is itself the child host of `janus.example.net`. We could specify this relationship in exactly the same way. Take a look at the following code:

```
define host {
    use          linux-router
    host_name    calpe.example.net
    alias        calpe
    address      192.0.2.129
    parents      janus.example.net
}
```

It's also possible to set multiple parents for a host if there are two possible paths to the same machine, as follows:

```
define host {
    use          linux-server
    host_name    corsica.example.net
    alias        corsica
    address      192.0.2.71
    parents      calpe.example.net, janus.example.net
}
```

With this configuration, `corsica.example.net` would only be deemed UNREACHABLE if both of its parent hosts were down. This kind of configuration is useful to account for redundant paths in a network—for example, spanning tree technologies or BGP/OSPF

failover.

After you set up a good basic structure for your network using the `parents` directive, definitely check out the *Using the network map* recipe in this chapter to get some automatic visual feedback about your network's structure as generated through your new configuration.

## See also

- The *Using the network map*, section from [Chapter 8](#), *Managing Network Layout*
- The *Establishing a host dependency*, section from [Chapter 8](#), *Managing Network Layout*
- The *Specifying which states to be notified about*, section from [Chapter 4](#), *Configuring Notifications*
- The *Configure notification groups*, section from [Chapter 4](#), *Configuring Notifications*



# Using the network map

In this recipe, we'll build on having configured a simple network hierarchy as described in the *Creating a network host hierarchy* recipe in this chapter by examining this hierarchy in the **network map** (or **status map**) in the Nagios Core web interface. The network map takes the form of a generated graphic showing the hierarchy of hosts and their current states. It allows filtering to show specific hosts and clicking hosts to navigate through larger networks.



# Getting ready

You will need to run Nagios Core 4.0 or a newer server and have access to its web interface. You will also need permission to view the states of hosts, preferably all hosts. You can arrange this by adding your username in the `authorized_for_all_hosts` directive, normally in `/usr/local/nagios/etc/cgi.cfg`; for example, for the user `tom`, you can execute the following:

```
authorized_for_all_hosts=nagiosadmin,tom
```

By default, the `nagiosadmin` user should have all the necessary permissions to view the complete map.

The network map is not particularly useful without at least a few hosts configured and arranged in a hierarchy; so, if you have not set any parents directives for your hosts, you may wish to read the *Creating a network host hierarchy* recipe in this chapter first and arranging your monitored hosts as it explains.

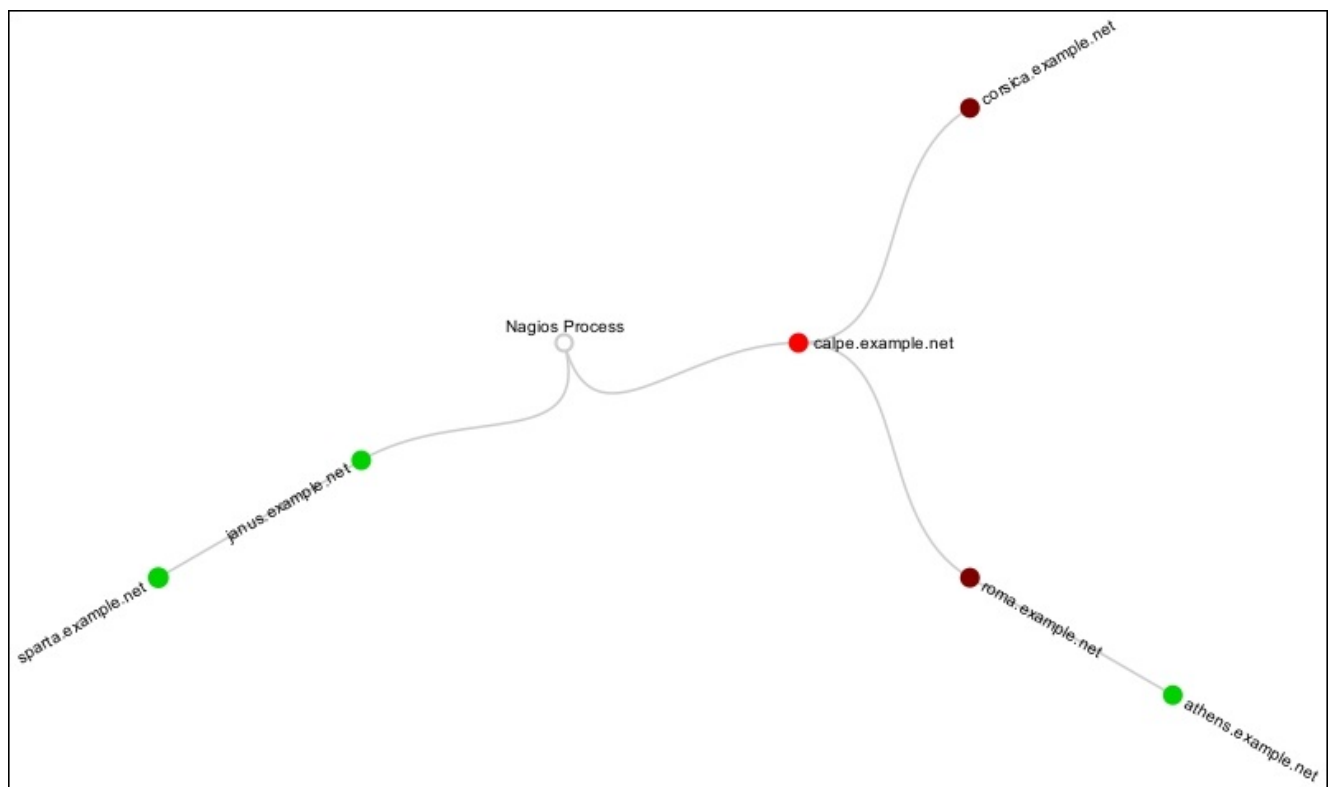
# How to do it...

We can inspect the network map for our newly configured host hierarchy like so:

1. First, log in to the Nagios Core web interface.
2. Click on the **Map** item in the left-hand side menu.



3. You should be presented with a generated graphic showing all the hosts in your network that your user has permission to view similar to the following:

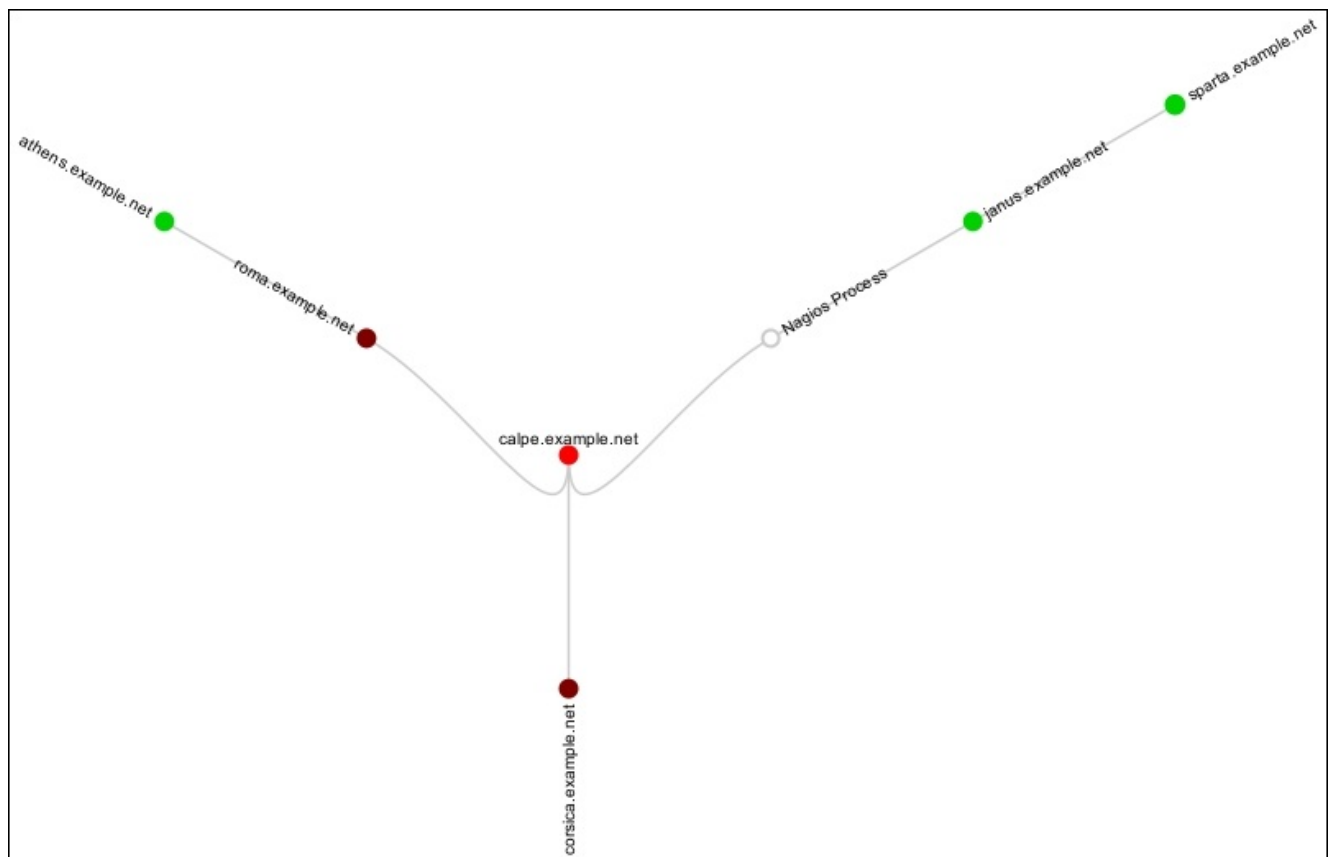


4. Let the mouse hover over any host to see a panel breaking down the host's current state, as follows:

Name: **janus.example.net**  
Alias: **janus.example.net**  
Address: **192.0.2.128**  
State: **up**  
State Duration: **16713d 2h 34m 28s**  
Last Status Check: **Mon Oct 05 15:31:21 2015**  
Last State Change: **unknown**  
Parent Host(s): **None (This is a root host)**  
Immediate Child Host(s): **1**  
Services:

Click node to collapse/expand its children  
Shift-click node to make it the root

- By default, the network map is centered around the Nagios Process icon. Try holding down *Shift* and clicking on one of your hosts to recenter the map—in this example, on `calpe.example.net`, as follows:



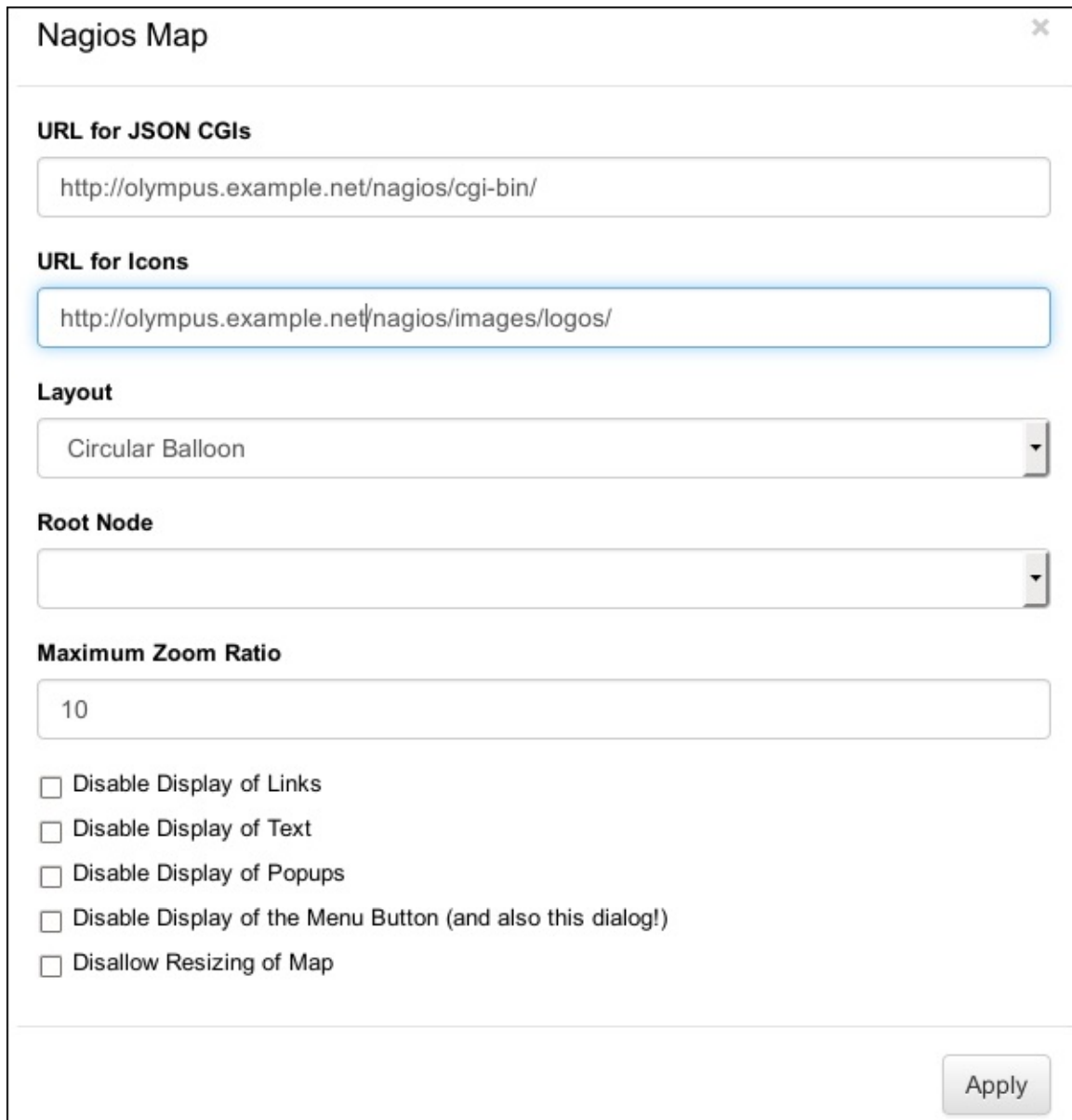
## How it works...

The network map is automatically generated from your host configuration. By default, it arranges the hosts outward from the central Nagios Process icon using lines to show dependencies and adjusting their colors to green for UP states and red for DOWN or UNREACHABLE states.

This map is generated via AngularJS, a JavaScript library. Legacy Map from older versions of Nagios Core, available with the **(Legacy)** link in the left-hand side menu, is generated with the GD2 library and written by Thomas Boutell.

# There's more...

Note that the form available by clicking on the icon in the upper right-hand side allows you to customize the appearance of the map directly. Take a look at the following screenshot:



The screenshot shows a dialog box titled "Nagios Map" with a close button (X) in the top right corner. The dialog contains several configuration fields and checkboxes:

- URL for JSON CGIs**: A text input field containing "http://olympus.example.net/nagios/cgi-bin/".
- URL for Icons**: A text input field containing "http://olympus.example.net/nagios/images/logos/". This field is highlighted with a blue border.
- Layout**: A drop-down menu currently showing "Circular Balloon".
- Root Node**: An empty drop-down menu.
- Maximum Zoom Ratio**: A text input field containing "10".
- Checkboxes**: Five unchecked checkboxes are listed at the bottom:
  - ☐ Disable Display of Links
  - ☐ Disable Display of Text
  - ☐ Disable Display of Popups
  - ☐ Disable Display of the Menu Button (and also this dialog!)
  - ☐ Disallow Resizing of Map
- Apply Button**: A button labeled "Apply" is located in the bottom right corner.

Of particular interest in this screen is the **Layout** drop-down menu, which can be adjusted to try several different methods of displaying the network layout.

After selecting or changing any one of these options, you will need to click on **Apply** to make the changes appear on the map.

The appearance of the status map can be configured well beyond this by changing directives in the Nagios Core configuration file and adding some directives to your hosts; take a look at the recipes under *See also* for some examples of how this is done.

## See also

- The *Customizing appearance of the network map* recipe in this chapter
- The *Choosing icons for hosts* recipe in this chapter
- The *Specifying coordinates for a host on the network map* recipe in this chapter
- The *Using the network map as an overlay* recipe in this chapter



# Choosing icons for hosts

In this recipe, you'll learn how to select graphics for hosts to appear in various parts of the Nagios Core web interface. This is done by adding directives to a host to specify the paths to appropriate images to represent it.

Adding these definitions has no effect on Nagios Core's monitoring behavior; they are mostly cosmetic changes, although it's useful to see at a glance whether a particular node is a server or a workstation, particularly on the network map.



# Getting ready

You will need to run Nagios Core 4.0 or a newer server and have access to its web interface. You must also be able to edit the configuration files for the server.

It's a good idea to verify that you actually have the required images installed. The default set of icons is included in `/usr/local/nagios/share/images/logos`. Don't confuse this with the directory above it, `images`, which contains images used as part of the Nagios Core web interface itself.

In the `logos` directory, you should find a number of images in various formats under it. In this example, we're interested in the router and rack-server icons:

```
$ ls /usr/local/nagios/share/images/logos/{router,rack-server}.*  
/usr/local/nagios/share/images/logos/rack-server.gd2  
/usr/local/nagios/share/images/logos/rack-server.gif  
/usr/local/nagios/share/images/logos/router.gd2  
/usr/local/nagios/share/images/logos/router.gif
```

To get the full benefit of the icons, you'll likely want to be familiar with using the network map and have access to view it with the appropriate hosts in your own Nagios Core instance. The network map is introduced in the *Using the network map* recipe in this chapter.

# How to do it...

We can define the images to be used in displaying our host like so:

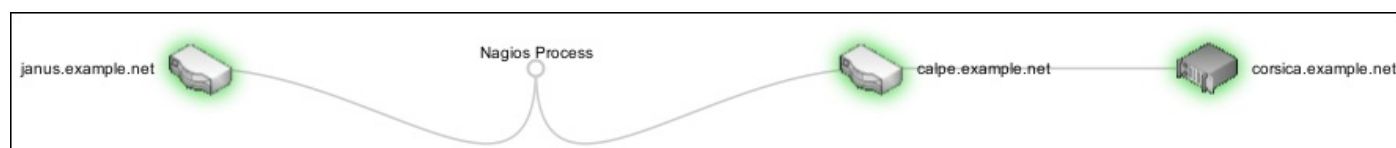
1. Change to the objects' configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to its directory instead and run the following line:  
  
**# cd /usr/local/nagios/etc/objects**
2. Add three new directives to each of the hosts to which you want to apply the icons. In this example, the rack-server icon is assigned to `corsica.example.net` and the router icon to both `calpe.example.net` and `janus.example.net`, as follows:

```
define host {
    use                linux-server
    host_name          corsica.example.net
    alias              corsica
    address            192.0.2.71
    icon_image         rack-server.gif
    icon_image_alt     Rack Server
    statusmap_image    rack-server.gd2
}
define host {
    use                linux-router
    host_name          janus.example.net
    alias              janus
    address            192.0.2.128
    icon_image         router.gif
    icon_image_alt     Router
    statusmap_image    router.gd2
}
define host {
    use                linux-router
    host_name          calpe.example.net
    alias              calpe
    address            192.0.2.129
    icon_image         router.gif
    icon_image_alt     Router
    statusmap_image    router.gd2
}
```

3. Validate the configuration and restart the Nagios Core server via the following code:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, a visit to the status map should show the appropriate hosts now being displayed with icons rather than small circles.



The **Hosts** list should also include a scaled-down version of the image, as in the following screenshot:

| Host  |   | Status  | Last Check  |
|----------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| calpe.example.net                                                                      |   | UP                                                                                       | 10-05-2015 16:08:12                                                                          |
| corsica.example.net                                                                    |   | UP                                                                                       | 10-05-2015 16:08:12                                                                          |
| janus.example.net                                                                      |   | UP                                                                                       | 10-05-2015 16:08:12                                                                          |

## How it works...

When a host list, service list, or network status map is generated, it checks for the presence of `icon_image` or `statusmap_image` values for each host object, reads the appropriate image if defined, and includes it as part of its processing. The network status map defaults to displaying only a colored circle in the absence of a value for the `statusmap_image` directive.

Note that for the `statusmap_image` directive, we chose the `.gd2` version of the icon rather than the `.gif` version. This applies to the legacy version of the status map for performance reasons; this status map is generated with the GD2 library, which deals more efficiently with its native `.gd2` format.

The `icon_image_alt` directive defines the value for the `alt` attribute when the image is displayed in an `<img>` HTML tag. Most web browsers will show the contents of this tag after briefly hovering over the icon.

## There's more...

If you have a number of hosts that need to share the same image, it is good practice to inherit from a common host template with the appropriate directives set. For our example, we might define a template like so:

```
define host {
    name                router-iconicon_image        router.gif
    icon_image_alt      Router
    statusmap_image     router.gd2
    register            0}
```

We could then apply the image settings directly to both our routers simply by inheriting from this template and by adding it to the use directive. Take a look at the following code:

```
define host {
    use                linux-router,router-icon
    host_name          janus.example.net
    alias              janus
    address            192.0.2.128
}
define host {
    use                linux-router,router-icon
    host_name          calpe.example.net
    alias              calpe
    address            192.0.2.129
}
```

If you don't like the included icon set, there are many icon sets available online on the Nagios Exchange site at <https://exchange.nagios.org/>. If you want, you could even make your own out of physical pictures of your hardware saved in standard PNG or GD2 format.

## See also

- The *Using the network map* recipe in this chapter
- The *Specifying coordinates for a host on the network map* recipe in this chapter



# Establishing a host dependency

In this recipe, you'll learn how to establish a **host dependency** between two hosts. This feature can be used to control how Nagios Core checks hosts and notifies them about problems in situations where if one host is `DOWN`, it implies that at least one other host is necessarily `DOWN`.



# Getting ready

First of all, it's very important to note that this is not quite the same thing as a host being UNREACHABLE, which is what the parents directive is for, as discussed in the *Creating a network host hierarchy* recipe in this chapter. Most of the time, a host actually being DOWN does not mean that other hosts actually go DOWN by definition. It's more typical for a child host to simply be UNREACHABLE; it might work fine, but Nagios Core can't verify this because of the DOWN host in its path.

However, there's one particularly broad category where host dependencies are definitely useful: the host/guest relationship of virtual machines. If you monitor both a **host** physical machine and one or more **guest** virtual machines, then the virtual machines will definitely be dependent on the host; if the host machine is actually in the DOWN state and has no redundant failover, then it would imply that the guests were DOWN as well and not simply UNREACHABLE.

We'll use virtualization as an example with two virtual machines, `zeus.example.net` and `athena.example.net`, running on a host machine, `ephesus.example.net`. All three are already monitored, but we'll establish a host dependency so that Nagios Core doesn't notify anyone about the guests' state if it determines that the host is DOWN.

You will need Nagios Core 4.0 or a newer server and to have shell access to change its backend configuration.

# How to do it...

We can establish our host dependencies like so:

1. Change to the objects' configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to its directory instead and run the following line of code:

```
# cd /usr/local/nagios/etc/objects
```

2. Create or edit an appropriate file that will be included by the configuration in `/usr/local/nagios/etc/nagios.cfg`. A sensible choice could be `/usr/local/nagios/etc/objects/dependencies.cfg`, as follows:

```
# vi dependencies.cfg
```

3. Now, add a hostdependency definition. In our case, the definition looks similar to the following. Note that you can include multiple dependent hosts by separating their names with commas, as follows:

```
define hostdependency {
    host_name                ephesus.example.net
    dependent_host_name      zeus.example.net, athena.example.net
    execution_failure_criteria  n
    notification_failure_criteria d,u
}
```

4. Edit `nagios.cfg` to include a reference to this new file so that it gets included in the configuration via the following code:

```
cfg_file=/usr/local/nagios/etc/objects/dependencies.cfg
```

5. Validate the configuration and restart the Nagios Core server by executing this code:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, if the `ephesus.example.net` host goes down and takes both the `zeus.example.net` and `athena.example.net` hosts down with it, checks to all three hosts will continue, but notifications will be suppressed for the two guest hosts.

# How it works...

The host dependency object's four directives are as follows:

- `host_name`: This is the name of the host on which at least one other host is dependent. This can also be a comma-separated list of host names.
- `dependent_host_name`: This is the name of the dependent host. Again, this can be a comma-separated list.
- `execution_failure_criteria`: This defines a list of states for the host being depended upon. If this host is in any of these states, then Nagios Core will skip the checks for the dependent hosts. This can be a comma-separated list of any of the following flags:
  - `o`: The depended-upon host is UP
  - `d`: The depended-upon host is DOWN
  - `u`: The depended-upon host is UNREACHABLE
  - `p`: The depended-upon host is PENDING (that is, not checked yet)

Alternatively, the single `n` flag can be used (as it is in this example) to specify that the checks should take place regardless of the depended-upon host's state.

- `notification_failure_criteria`: This defines a list of states for the host being depended upon. If this host is in any of these states, then notifications for the dependent host will not be sent. The flags are the same as for `execution_failure_criteria`; in this example, we chose to suppress the notifications if the host being depended upon is DOWN or UNREACHABLE.

When Nagios Core notices that the `zeus.example.net` or `athena.example.net` hosts have apparently gone to the DOWN state as a result of a failed host check, it refers to its configuration to check whether there are any dependencies for the host and finds that they depend on `ephesus.example.net`.

It then checks the status of `ephesus.example.net` and finds it to be DOWN. Referring to `execution_failure_criteria` and finding `n`, it continues to run checks for both of the dependent hosts as normal. However, referring to `notification_failure_criteria` and finding `d`, `u`, it determines that notifications should be suppressed until the host returns to an UP state.

## There's more...

We can specify groups rather than host names using the `hostgroup_name` and `dependent_hostgroup_name` directives, as follows:

```
define hostdependency {
    hostgroup_name          vm-hosts
    dependent_hostgroup_name vm-guests
    execution_failure_criteria    n
    notification_failure_criteria d,u
}
```

We can also provide comma-separated lists of hosts on which to depend, as shown in the following code:

```
define hostdependency {
    host_name          ephesus.example.net,alexandria.example.net
    dependent_host_name zeus.example.net,athena.example.net
    execution_failure_criteria    n
    notification_failure_criteria d,u
}
```

If a host depends on more than one host, the check or notification rules apply if any of its dependencies are not met rather than all of them. For the preceding example, this means that if `ephesus.example.net` was in the `DOWN` state but `alexandria.example.net` was in `UP`, the dependency would still suppress checks or notifications for all the dependent hosts.

This means that host dependencies are not really suitable in redundant scenarios where the loss of one of the depended-upon hosts does not necessarily imply the loss of all its dependent hosts. You are likely to find that monitoring nodes as a cluster is a better fit for this situation; this is discussed in the *Monitoring individual nodes as a cluster* recipe, which is also in this chapter.

## See also

- The *Establishing a service dependency* recipe in this chapter
- The *Creating a network host hierarchy* recipe in this chapter
- The *Monitoring individual nodes as a cluster* recipe in this chapter



# Establishing a service dependency

In this recipe, you'll learn how to establish a service dependency between two services. This feature can be used to control how Nagios Core checks for hosts and notifies about problems in situations where if one host is in a `PROBLEM` state, it implies that at least one other service is necessarily also in the `PROBLEM` state.

# Getting ready

You will need Nagios Core 4.0 or a newer server and to have shell access to change its backend configuration. You will also need to have at least two services defined, one of which is by definition dependent on the other; this means that if the depended-upon service were to enter CRITICAL state, then it would imply that the dependent service would also be CRITICAL.

We'll use a simple example: let's suppose that we are testing authentication to a mailserver `marathon.example.net` with a service `MAIL_LOGIN` and also checking a database service `MAIL_DB` on the same host that stores the login usernames and password hashes.

In this situation, it might well be the case that if `MAIL_DB` is not working, then `MAIL_LOGIN` will almost certainly not be working either. If so, then we can configure Nagios Core to be aware that the `MAIL_LOGIN` service is **dependent** on the `MAIL_DB` service.



# How to do it...

We can establish our service dependency like so:

1. Change to the objects' configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to its directory instead via the following code:
2. Create or edit an appropriate file that will be included by the configuration in `/usr/local/nagios/etc/nagios.cfg`. A sensible choice could be `/usr/local/nagios/etc/objects/dependencies.cfg`.
3. Add a servicedependency definition. In our case, the definition looks similar to this:

```
define servicedependency {  
    host_name                marathon.example.net  
    service_description      MAIL_DB  
    dependent_host_name      marathon.example.net  
  
    dependent_service_description MAIL_LOGIN  
    execution_failure_criteria c  
    notification_failure_criteria c  
}
```

4. Edit `nagios.cfg` to include a reference to this new file so that it's included in the configuration, as follows:

```
cfg_file=/usr/local/nagios/etc/objects/dependencies.cfg
```

5. Validate the configuration and restart the Nagios Core server by executing the following:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios restart
```

With this done, if the `MAIL_DB` service fails for whatever reason and enters the `CRITICAL` state, the `MAIL_LOGIN` service will skip its checks of this service and also any notifications that it would normally send about its own problems. Note that the web interface may still show it scheduling checks, but it won't actually run them.

# How it works...

The service dependency object's five directives are as follows:

- `host_name`: This is the name of the host with which these services are associated.
- `service_description`: This is the description of the service being depended upon. This can be a comma-separated list.
- `dependent_service_description`: This is the description of the dependent service. This can also be a comma-separated list.
- `execution_failure_criteria`: This defines a list of states for the service being depended upon. If this service is in any of these states, then Nagios Core will skip checks for the dependent services. This can be a comma-separated list of any of the following flags:
  - `o`: The depended-upon service is OK
  - `w`: The depended-upon service is WARNING
  - `c`: The depended-upon service is CRITICAL (as in this example)
  - `u`: The depended-upon service is UNKNOWN
  - `p`: The depended-upon service is PENDING (that is, not checked yet)

Alternatively, the single `n` flag can be used to specify that the checks should take place regardless of the depended-upon service's state. In this example, we chose the `c` value, to suppress service checks only if the service being depended upon is CRITICAL.

- `notification_failure_criteria`: This defines a list of states for the service being depended upon. If this service is in any of these states, then notifications for the dependent service will not be sent. The flags are the same as for `execution_failure_criteria`; in this example, we again chose the `c` value to suppress the notifications only if the service being depended upon is CRITICAL.

When Nagios Core notices that the `MAIL_DB` service has gone into the CRITICAL state as a result of a failed service check, it refers to its configuration to check whether there are any dependencies for the service and finds that they depend on `MAIL_LOGIN`.

It then checks the status of `MAIL_DB` and finds it to be CRITICAL. Referring to `execution_failure_criteria` and finding `c`, it prevents checking for both of the dependent services. Referring to `notification_failure_criteria` and also finding `c`, it decides that notifications should be suppressed until the service returns to any other state.

## There's more...

Note that services do not have to be on the same host to depend upon one another. We can add `dependent_host_name` or `dependent_hostgroup_name` to specify other hosts, as follows:

```
define servicedependency {  
    host_name                marathon.example.net  
    service_description      MAIL_DB  
    dependent_host_name      sparta.example.net  
    dependent_service_description WEBMAIL_LOGIN  
    execution_failure_criteria c  
    notification_failure_criteria c  
}
```

In this example, the `WEBMAIL_LOGIN` service on `sparta.example.net` is defined as dependent on the `MAIL_DB` service on `marathon.example.net`. Note that the values for `host_name` and `dependent_host_name` are different.

## See also

- The *Establishing a host dependency* recipe in this chapter
- The *Monitoring individual nodes as a cluster* recipe in this chapter



# Monitoring individual nodes in a cluster

In this recipe, you'll learn how to monitor a collection of hosts in a cluster using the `check_cluster` plugin included in the Nagios Plugins standard. Being able to monitor more than one host collectively is useful in situations with redundancy; one of a set of hosts being in the `DOWN` state, perhaps for power conservation or maintenance reasons, is not necessarily a cause for notification. However, if a larger number or all of the hosts were down, we would definitely want to be notified. Using `check_cluster` allows us to arrange this.

# Getting ready

You will need Nagios Core 4.0 or a newer server and to have shell access to change its backend configuration. You will also need to have at least two monitored hosts in a redundant setup for some function, such as database replication, nameservers, or load-balanced web servers.

You should also be familiar with the way hosts and services are defined, writing commands in particular; these concepts are discussed in [Chapter 1](#), *Understanding Hosts, Services, and Contacts* of this book.

For this example, we'll work with three blade servers with the hostnames `achilles.example.net`, `odysseus.example.net`, and `agamemnon.example.net` running in a redundant cluster to support a virtual hosting environment. All three hosts are already monitored to send an e-mail message if one of them goes down. We will arrange a `check_cluster` service on a dummy host in such a way that:

- If none of the blades is down, the service is OK
- If one of the blades is down, the service enters the WARNING state, again notifying as appropriate
- If two or all three of the blades are down, the service enters the CRITICAL state, again notifying as appropriate

# How to do it...

We can arrange a cluster check for our hosts like so:

1. Change to the objects' configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your host in a different file, move to its directory instead and run the following line:

```
# cd /usr/local/nagios/etc/objects
```

2. Create or edit an appropriate file to define a new command. A sensible choice might be `/usr/local/nagios/etc/objects/commands.cfg`.
3. Define two new commands in this file—`check_dummy` and `check_host_cluster`—as follows:

```
define command {
    command_name    check_dummy
    command_line    $USER1$/check_dummy $ARG1$ $ARG2$
}
define command {
    command_name    check_host_cluster
    command_line    $USER1$/check_cluster -h -d $ARG1$ -w $ARG2$ -c
$ARG3$
}
```

4. Create or edit an appropriate file that will be included by the configuration in `/usr/local/nagios/etc/nagios.cfg`. A sensible choice might be `/usr/local/nagios/etc/objects/clusters.cfg`.
5. Define a dummy host for the cluster with the values shown in the following code:

```
define host {
    use                generic-host
    host_name          example-blade-cluster
    alias              Example.net Blade Cluster
    address            127.0.0.1
    max_check_attempts 1
    contact_groups     admins
    check_command       check_dummy!0!"Dummy host only"
}
```

Note that the address directive has the value `127.0.0.1`; this is deliberate as the dummy host itself does not really need to be actively checked or send any notifications.

6. Now, add a service for the dummy host via the following lines:

```
define service {
    use                generic-service
    host_name          example-blade-cluster
    service_description CLUSTER
    check_command
check_host_cluster!$HOSTSTATEID:achilles.example.net$, $HOSTSTATEID:odys
seus.example.net$, $HOSTSTATEID:agamemnon.example.net$!@1:!@2:
    notification_options c,w,r
```



}

The suggestion of inheriting from `generic-service` is only an example; you will likely want to use your own template or values. Note that the value for `check_command` is all on one line with no spaces. You should substitute this with the hostnames of your own machines.

7. Add a reference to the new file to `nagios.cfg` so that it's included in the configuration, as follows:

```
cfg_file=/usr/local/nagios/etc/objects/clusters.cfg
```

8. Next, validate the configuration and restart the Nagios Core server via the following script:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, the `CLUSTER` service on the `example-blade-cluster` dummy host should be available for viewing and will change the state and send notifications the same way as any other service if the hosts in the cluster come up or go down.

|                       |         |          |                     |             |     |                                                             |
|-----------------------|---------|----------|---------------------|-------------|-----|-------------------------------------------------------------|
| example-blade-cluster | CLUSTER | OK       | 10-05-2015 18:22:06 | 0d 0h 0m 7s | 1/3 | CLUSTER OK: Host cluster: 3 up, 0 down, 0 unreachable       |
| example-blade-cluster | CLUSTER | WARNING  | 10-05-2015 18:26:41 | 0d 0h 0m 2s | 1/3 | CLUSTER WARNING: Host cluster: 2 up, 1 down, 0 unreachable  |
| example-blade-cluster | CLUSTER | CRITICAL | 10-05-2015 18:31:50 | 0d 0h 0m 8s | 3/3 | CLUSTER CRITICAL: Host cluster: 1 up, 2 down, 0 unreachable |

## How it works...

The host added in this recipe is just a hook on which to hang the `CLUSTER` service, which performs the actual check. This is why we used a command using the `check_dummy` plugin to always return an OK state. Take a look at the following code:

```
# /usr/local/nagios/libexec/check_dummy 0 "Dummy host only"
OK: Dummy host only
```

The `check_cluster` command is actually quite simple. It performs no actual checks of its own; instead, it determines the states based on the current state of other hosts or services.

This is why the `$HOSTSTATEID:hostname$` macros are used in the `check_command` directive for the service; they evaluate to a number indicating the state of the host with the hostname specified after the colon.

For example, if `achilles.example.net` and `odysseus.example.net` were in the `UP` state but `agamemnon.example.net` were in `DOWN`, the command run by Nagios Core for the check would look similar to this after the macros were expanded:

```
/usr/local/nagios/libexec/check_cluster -h -d 0,0,1 -w @1: -c @2:
```

We can run this ourselves as the `nagios` user to check the output, as follows:

```
# sudo -s -u nagios
$ /usr/local/nagios/libexec/check_cluster -h -d 0,0,1 -w @1: -c @2:
CLUSTER WARNING: Service cluster: 2 ok, 1 warning, 0 unknown, 0 critical
```

The comma-separated states given in the `-d` option to the plugin correspond to two hosts being in the `UP` state (which is the state ID of `0`) and one host being in `DOWN` (which is the state ID of `1`). The `-w` option's value of `@1:` means that a `WARNING` state will be entered if one or more of the hosts is down. Similarly, the `-c` option's value of `@2:` means that the `CRITICAL` state will be entered into if two or more of the hosts go down.

This allows you to customize notifications to be sent based on the number of hosts in the `DOWN` state rather than merely monitoring the hosts individually. Once you're confident that this is working correctly, you may even choose to prevent the individual hosts from sending notifications to your pager and have the `CLUSTER` service notify instead.

## There's more...

If you have a cluster of services rather than hosts to monitor, this can be done using the `-s` option to perform `check_cluster` rather than the `-h` option. In this case, instead of the `$HOSTSTATEID:<host_name>$` macros, you would use the `$SERVICESTATEID:<host_name>:<service_description>$` macros. An example configuration might look something similar to this for a cluster of two web servers—`sparta.example.net` and `athens.example.net`—given a dummy host named `example.net-http-cluster`:

```
define command {
    command_name    check_service_cluster
    command_line    $USER1$/check_cluster -s -d $ARG1$ -w $ARG2$ -c $ARG3$
}
define service {
    use              generic-servicehost_name                example.net-
http-clusterservice_description    CLUSTER_HTTP
    check_command    check_service_cluster!$SERVICESTATEID:sparta.example.net:HTTP$, $SERVICESTATEID:athens.example.net:HTTP$!@1:!@2:
}
```

## See also

- The *Establishing a host dependency* recipe in this chapter
- The *Establishing a service dependency* recipe in this chapter



# Using the network map as an overlay

In this recipe, you'll learn how to use a background for the legacy network map and deliberate the placement of hosts in specific points on it to make a kind of network status weathermap to see host statuses at a glance in a geographical context. At the time of writing, this method only works for the legacy map, which is available by clicking on **(Legacy)** to the right of the Map entry in the left-hand side menu.

# Getting ready

You will need Nagios Core 4.0 or a newer server and have shell access to change its backend configuration. You should also have at least a couple of hosts configured to place on the map and understand the basics of using the Nagios network map and icons for hosts. These are discussed in the *Using the network map* and *Choosing icons for hosts* recipes, both in this chapter.

You should also select a background image on which you can meaningfully place hosts. If you are monitoring an office network, this could be a floor plan of the building or server room. If you're monitoring a nationwide Internet service provider, you could use a map of your state or country. Some administrators even like to use pictures of physical equipment and place the Nagios Core hosts over their physical analogues. In this example, we'll use a map of Australia, 640 by 509 pixels in size, which is a public domain image retrieved from the Natural Earth website at <http://www.naturalearthdata.com/>.



The background image can be anything you like and several graphics formats, including PNG, can be used. However, for the sake of quick map rendering, it's recommended to use an image in the GD2 file format with the .gd2 extension. If you have your image in the PNG format, you can generate a GD2 image from it using the free tool `pngtogd2`, as follows:

```
$ pngtogd2 australia.png australia.gd2 0 1
```

This tool is available on Debian-derived systems in the `libgd-tools` package. Its source code is also available online at <http://www.libgd.org/>.

# How to do it...

We can set up a background for our network map like so:

1. Copy your GD2 format image into the images subdirectory of the `physical_html_path` directory. In the default installation, this is `/usr/local/nagios/share/images`; if it is different, you can find the definition of `physical_html_path` in `/usr/local/nagios/etc/cgi.cfg`. Run the following code:

```
# cp /home/tom/australia.gd2 /usr/local/nagios/share/images
```

2. Change to the configuration directory for Nagios Core. In the default installation, this is `/usr/local/nagios/etc`. Edit the `cgi.cfg` file, via the following:

```
# cd /usr/local/nagios/etc
# vi cgi.cfg
```

3. Look for the `statusmap_background_image` directive in this file. Uncomment it and make its value the name of your image via the following:

```
statusmap_background_image=australia.gd2
```

4. Look for the `default_statusmap_layout` directive in the same file. Change it to `0`, which corresponds to the **User-defined coordinates** layout, as follows:

```
default_statusmap_layout=0
```

5. Change to the objects' configuration directory for Nagios Core. In the default installation, this is `/usr/local/nagios/etc/objects`. In this example, the Australian hosts are in a file called `australia.cfg`.

6. Add `2d_coords` directives to each of the hosts you want to display on the map. You might like to include definitions for `statusmap_image` here too, as done here:

```
# vi australia.cfg
define host {
    use                linux-server
    host_name          adelaide.example.net
    address            192.0.2.140
    2d_coords          390,360
    statusmap_image    rack-server.gd2
}
define host {
    use                linux-server
    host_name          cairns.example.net
    address            192.0.2.141
    2d_coords          495,100
    statusmap_image    rack-server.gd2
}
... etc...
```

7. If a new file were created for the hosts, include a reference to it in `nagios.cfg`, as follows:

```
cfg_file=/usr/local/nagios/etc/objects/australia.cfg
```



8. For the `2d_coords` directive, supply two comma-separated values describing the coordinates for the placement of the host. For example, `adelaide.example.net` is 390 pixels from the left-hand side and 360 pixels from the top. A convenient way to get the coordinates is using GIMP, the open-source imaging tool, or even a simple tool such as MS Paint; load the image and let the cursor hover over the point you wish to use to find its pixel coordinates.

Validate the configuration and restart the Nagios Core server, as follows:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, upon visiting the network map in the Nagios Core web interface by clicking on **Map** in the left-hand side menu, your hosts will be placed in their corresponding positions on the map, including the normal lines and colors to specify child-parent relationships and reachability, as shown in the following screenshot:

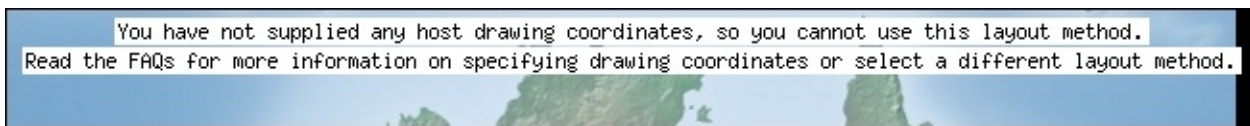


## How it works...

The `default_statusmap_layout` directive fixes the network map into the **User-supplied coords mode** by default. In this mode, only the hosts with values for `2d_coords` are shown, and they are displayed at fixed points on the map rather than being dynamically placed.

It's possible to use this display mode without a background if we wish, but we can give a lot of useful context to the picture of the network generated by taking the extra step of using an actual background image.

Note that if you don't have any hosts with coordinates defined, you'll receive an error that looks something similar to this:



## There's more...

Using the network map with an image background can be particularly helpful to see not only the statuses of individual hosts at a glance, but in the case of outages from multiple hosts, also looking for possible geographical causes. If all of the nodes in one part of the city or country went down at once, we would be able to see this at a glance. This makes the network map an excellent choice for a network monitoring display or as one of the first ports of call in diagnosing large-scale problems.

The network map is very useful in this way and, graphically, it is probably the most impressive part of the Nagios Core web interface. If you would like even more options and an impressive range of visualizations for host statuses, you may like to consider looking at the excellent **NagVis** extension, which could fill a whole book in itself. There is a brief introduction to its usage in the *Getting extra visualizations with NagVis* recipe in [Chapter 11](#), *Automating and Extending Nagios Core*.

## See also

- The *Creating a network host hierarchy* recipe in this chapter
- The *Using the network map* recipe in this chapter
- The *Choosing icons for hosts* recipe in this chapter
- The *Getting extra visualizations with NagVis* section from [Chapter 11](#), *Automating and Extending Nagios Core*



# Chapter 9. Managing Configuration

In this chapter, we will cover:

- Grouping configuration files in directories
- Keeping a configuration under version control
- Configuring host roles using groups
- Building groups using regular expressions
- Using inheritance to simplify configuration
- Defining macros in a resource file
- Using another object's directives in a host or service check
- Using custom directives
- Dynamically building host definitions

# Introduction

A major downside of Nagios Core's configuration being so flexible is that without proper management, the configuration can easily balloon into hundreds of files with thousands of objects, all having unclear dependencies. This can be frustrating when you attempt to make significant changes to a configuration or even for something as simple as removing a host or sifting through dependencies to find what's causing errors in the configuration that prevents you from restarting.

It's therefore important to build your configuration carefully using as much **abstraction** as possible to allow adding, changing, and removing hosts and service definitions from the configuration painlessly and to avoid a duplication of configuration. Nagios Core provides a few ways of dealing with this, most notably in the judicious use of **groups** and **templates** for fundamental objects. Duplication of network-specific and volatile data, such as passwords, is also to be avoided and this is best done with the use of custom macros defined in a resource file.

This chapter's recipes will run through some examples of good practice for the configuration of a large network. The most important recipes are the first two: *Grouping configuration files in directories* and *Configuring host roles using groups*. If you're looking to untangle and revamp a messy configuration, these two recipes would be the best place to start.

At the end of the chapter, the final recipe, *Dynamically building host definitions*, will show one of the primary advantages of a tidy configuration in being able to easily generate configuration according to a list of hosts and services kept in some other external information source, such as a configuration management database (CMDB).





# Grouping configuration files in directories

In this recipe, you'll learn how to group configuration files in directories to greatly ease the management of configuration. We'll do this by configuring Nagios Core to load every file it can find ending with a `.cfg` extension in a given directory, which includes recursing through subdirectories. The end result will be that to have Nagios Core load a file, we only need to include it somewhere in this directory with an appropriate extension; we don't need to keep track of which files are being loaded and where, nor add any new definitions in `nagios.cfg`.

# Getting ready

You will need to have a server running Nagios Core 4.0 or later and access to the command line to change its configuration. You should be familiar with the loading of individual configuration files using the `cfg_file` directive in `/usr/local/nagios/etc/nagios.cfg`.

In particular, you should have a directory prepared that contains all of the configuration items you would like to be loaded by Nagios Core. In this example, we'll prepare a new directory called `/usr/local/nagios/etc/objects/hosts`, which will contain three configuration files, each defining the information for one host, as follows:

```
# ls -l /usr/local/nagios/etc/objects/hosts
athens.cfg
ithaca.cfg
sparta.cfg
```

You need to ensure that the directory and the files and subdirectories within it are all readable (though not necessarily owned) by the `nagios` user.

# How to do it...

We can arrange for Nagios Core to include all the files in a directory like so:

1. Change to the directory containing the `nagios.cfg` file. In the default installation, it is located at `/usr/local/nagios/etc/nagios.cfg`. Run the following line of code:

```
# vi nagios.cfg
```

2. Edit the file and, below any other `cfg_file` or `cfg_dir` directive, add the following line referring to the absolute path of the directory containing your `.cfg` files:

```
cfg_dir=/usr/local/nagios/etc/objects/hosts
```

## Tip

Note that this directive is `cfg_dir` rather than `cfg_file`.

3. Remove any `cfg_file` definition pointing to `.cfg` files in the new directory. This is to avoid loading the same objects twice, which will probably cause errors. In our example, we need to comment out our previous rules by loading these files individually:

```
#cfg_file=/usr/local/nagios/etc/objects/hosts/sparta.cfg  
#cfg_file=/usr/local/nagios/etc/objects/hosts/athens.cfg  
#cfg_file=/usr/local/nagios/etc/objects/hosts/ithaca.cfg
```

4. Validate the configuration and restart the Nagios Core server via the following line:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios restart
```

With this done, Nagios Core should have all of the files loaded with the `.cfg` extension found in the example directory or any of its subdirectories, saving us the burden of specifying them individually.

## How it works...

At load time, Nagios Core interprets the `cfg_dir` directive to mean that it should identify all of the `.cfg` files in a particular directory, including recursing through its subdirectories. It ignores files that do not have this extension, allowing us to include metadata or other file types (such as version control information directories) without causing problems.

As a result, defining a new host or service becomes as simple as adding the file in an appropriate included directory without needing to edit `nagios.cfg`.

## There's more...

It's important to note that this directive will include all the `.cfg` files in subdirectories as well. This might allow us to meaningfully group the configuration files thus:

```
$ find /usr/local/nagios/etc/objects
/usr/local/nagios/etc/objects/hosts
/usr/local/nagios/etc/objects/hosts
/usr/local/nagios/etc/objects/hosts/athens.cfg
/usr/local/nagios/etc/objects/hosts/sparta.cfg
/usr/local/nagios/etc/objects/hosts/ithaca.cfg
/usr/local/nagios/etc/objects/commands
/usr/local/nagios/etc/objects/commands/webserver-checks.cfg
/usr/local/nagios/etc/objects/services
/usr/local/nagios/etc/objects/services/webserver.cfg
```

For large networks, it's worth deciding on some suitable organizing principle for directories. One common approach is having a separate directory for hosts, services, and command definitions relevant to a particular DNS zone or larger subnet.

If you want to prevent a file from being included at any point, all you need to do is either move it out of the directory, or (more practically) rename it so that it no longer has a `.cfg` extension. One possibility is adding the word `.exclude`, as follows:

```
# mv unwanted-file.cfg unwanted-file.cfg.exclude
```

This will prevent Nagios Core from picking it up as part of its `cfg_dir`-searching algorithm.

## See also

- The *Using inheritance to simplify configuration* recipe in this chapter
- The *Keeping a configuration under version control* recipe in this chapter



# Keeping a configuration under version control

In this recipe, we'll place a Nagios Core configuration directory under version control in an attempt to keep track of the changes made to it and to enable us to reverse changes if there are problems.



# Getting ready

You should choose an appropriate version control system. The recipe will vary considerably depending on which system you use; there are too many options to demonstrate here, so we'll use the popular open-source content tracker Git, the basics of which are very easy to use for this kind of version control and do not require an external server. However, there's no reason you can't use Subversion or Mercurial if you prefer them. You should have the client for your chosen system (`git(1)`, `hg(1)`, `svn(1)`, and so on) installed on your server.

This will all work with any version of Nagios Core. It does not involve directly changing any part of the Nagios Core configuration, only keeping track of the files in it.

# How to do it...

We can place our configuration directory under version control like so:

1. Change to the root of the configuration directory. In the default installation, this is `/usr/local/nagios/etc`. Run the following line:

```
# cd /usr/local/nagios/etc
```

2. Run `git init` command to start a repository in the current directory as follows:

```
# git init
Initialized empty Git repository in /usr/local/nagios/etc/.git/.
```

3. Then, add all the files in the configuration directory with `git add` command as follows:

```
# git add .
```

4. Commit the files with `git commit` command via the following code:

```
# git commit -m "First commit of configuration"
[master (root-commit) 6a2c605] First commit of configuration.
 39 files changed, 3339 insertions(+), 0 deletions(-)
 create mode 100644 example/athens.example.net.cfg
 create mode 100644 example/ithaca.example.net.cfg
 create mode 100644 example/sparta.example.net.cfg
 ...
```

With this done, we should have a `.git` repository in `/usr/local/nagios/etc` tracking all the changes made to the configuration files.

# How it works...

Changes to the configuration files in this directory can now be reviewed for committing with `git status`. For example, if we changed the IP addresses of one of our hosts, we might be able to check the changes by typing the following:

```
# git status -s
M example/hosts/sparta.cfg
```

We could then commit this change with an explanatory message, as follows:

```
# git commit -am "Changed IP address of host"
[master d5116a6] Changed IP address of host
1 files changed, 1 insertions(+), 1 deletions(-)
```

We can review the change with `git log` via the following line of code:

```
# git log --oneline
d5116a6 Changed IP address of host
6a2c605 First commit of configuration
```

If we want to inspect exactly what was changed later on, we can use `git diff`, followed by the short commit ID given in the first column of the preceding output:

```
# git diff 6a2c605
diff --git a/example/hosts/sparta.cfg b/example/hosts/sparta.cfg
index 0bb3b0b..fb7c2a9 100644
--- a/example/hosts/sparta.cfg
+++ b/example/hosts/sparta.cfg
@@ -2,7 +2,7 @@ define host {
     use                linux-server
     host_name          sparta.example.net
     alias              sparta
-    address            192.0.2.101
+    address            192.0.2.102
 }
```

The full functionality of Git to manage these changes, including reverting to older revisions, is out of scope here. You can read more about how to use Git in general in Scott Chacon's excellent book entitled *Pro Git*, which is free to read online at <https://git-scm.com/book>.

## There's more...

Version control is particularly useful in this way when more than one person is editing a configuration because it allows us to determine who made a change and when. It also allows us to note the exact change set to review why it was changed and to undo or edit it if it causes problems.

If you use this method, it's a good idea to keep your configuration reasonably granular, using at least several files rather than just one or two. It will still work if you have three big files such as `hosts.cfg` or `services.cfg` for your network, but the differences between each commit will not be as clear. This is therefore a very good recipe to combine with *Grouping configuration files in directories*, which is also in this chapter.

Rather than merely the configuration directory, you may prefer to keep the entire Nagios Core directory under version control, plugins and all. This could be particularly handy if you upgrade your installation with new releases and want to see what's changed in your files in case it breaks anything. In this case, be careful to use your chosen version control system's "ignore" functionality to avoid tracking temporary or log files. For Git, take a look at the output of `git help ignore`.

## See also

- The *Grouping configuration files in directories* recipe in this chapter



# Configuring host roles using groups

In this recipe, you'll learn how to use the abstraction of host and service groups to your advantage in order to build a configuration where hosts and services can be added or removed more easily. We'll do this by defining **roles** for hosts using a hostgroup structure and then assigning relevant services to the hostgroup rather than to the hosts individually.

# Getting ready

You will need to have a server running Nagios Core 4.0 or later, have access to the command line to change its directories, and understand the basics of how hostgroups and servicegroups work. These are covered in the recipes *Creating a new hostgroup* and *Creating a new servicegroup* in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

In this example, we'll create two simple hostgroups; one will be called `servers`, for which a `PING` check should be made for its member hosts, and another will be called `webservers`, which should include `HTTP` checks for its member hosts. Once this is set up, we'll add an example host `sparta.example.net` to both groups, thereby easily assigning all the appropriate services to the host in one definition, which we can cleanly remove simply by deleting the host.



# How to do it...

We can create our group-based roles for hosts like so:

1. Change to the objects configuration directory. In the default installation, this is `/usr/local/nagios/etc/objects`. If you have already followed the *Grouping configuration files in directories* recipe in this chapter, your own directory may differ. Run the following line:

```
# cd /usr/local/nagios/etc/objects
```

2. In an appropriate file, perhaps `hostgroups.cfg`, define new hostgroups with names corresponding to the roles for the hosts. Do not assign them any members just yet. Execute the following code:

```
define hostgroup {
    hostgroup_name servers
    alias          Servers
}
define hostgroup {
    hostgroup_name web-servers
    alias          Web Servers
}
```

3. In a second appropriate file, perhaps `services.cfg`, define new services and assign them `hostgroup_name` values corresponding to the hostgroups added in the previous step, as follows:

```
define service {
    use                generic-service
    hostgroup_name     servers
    service_description PING
    check_command       check_ping!100,10%!200,20%
}
define service {
    use                generic-service
    hostgroup_name     web-servers
    service_description HTTP
    check_command       check_http
}
```

## Tip

Note that the use of the `generic-service` template is an example only; you will probably want to inherit from your own particular service template.

4. Add or edit your existing hosts to make them form part of the appropriate hostgroups using the `hostgroups` directive via the following script:

```
define host {
    use        linux-server
    host_name  sparta.example.net
    alias      sparta
    address    192.0.2.21
}
```

```

        hostgroups    servers,web-servers
    }
    define host {
        use            linux-server
        host_name      athens.example.net
        alias          athens
        address        192.0.2.22
        hostgroups    servers,web-servers
    }

```

5. If you already had services with the same `service_description` value as the ones you're adding in this recipe, you will need to remove them as this will possibly cause a conflict with the services added in the previous step.
6. Validate the configuration and restart the Nagios Core server, as follows:

```

# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart

```

With this done, you should now find that all of the services you defined for the hostgroups you created are attached to the appropriate hosts:

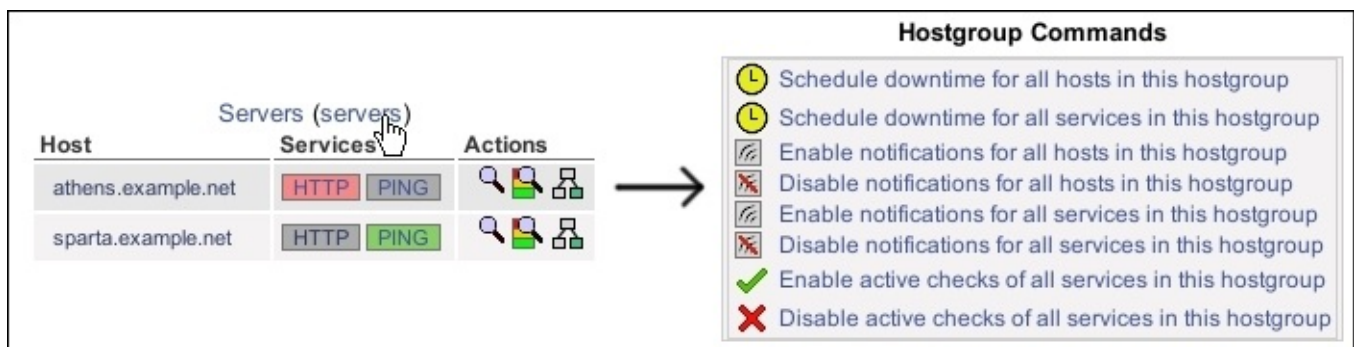
|                    |      |         |
|--------------------|------|---------|
| athens.example.net | HTTP | PENDING |
|                    | PING | PENDING |
| sparta.example.net | HTTP | PENDING |
|                    | PING | PENDING |

## How it works...

The configuration added in the preceding section avoids assigning services directly to hosts; instead, it assigns them to hostgroups, thereby creating roles for services that hosts can adopt or discard simply by becoming part of or leaving the group.

Apart from making the configuration shorter, another advantage to this approach is that if services are added this way, adding or deleting a host from the Nagios Core configuration requires nothing but adding or removing the host definition. Similarly, if a host takes on another role (for example, that of a web server adding a database functionality), we can modify the services being checked on it simply by modifying its hostgroups. This ends up being much easier than adding dependencies in other files each time we make a change to the functions of hosts.

Another advantage is that having hostgroups organized by a host function is helpful to apply batch operations such as scheduled downtime in one easy action or to check all the services for one particular type of host. Once a hostgroup is defined, we can run operations on all the hosts within it by clicking on its name in brackets in any of the hostgroup views.



The screenshot displays the Nagios Core web interface. On the left, a table lists hosts and their services. The 'Host' column shows 'athens.example.net' and 'sparta.example.net'. The 'Services' column shows 'HTTP' and 'PING' for each host. The 'Actions' column contains icons for search, refresh, and group actions. A hand cursor is hovering over the 'Servers (servers)' link. An arrow points from the 'Actions' column to a 'Hostgroup Commands' panel on the right. This panel lists various commands for the hostgroup, including scheduling downtime for hosts or services, enabling/disabling notifications, and enabling/disabling active checks.

| Host               | Services  | Actions                    |
|--------------------|-----------|----------------------------|
| athens.example.net | HTTP PING | [Search] [Refresh] [Group] |
| sparta.example.net | HTTP PING | [Search] [Refresh] [Group] |

**Hostgroup Commands**

- [Clock] Schedule downtime for all hosts in this hostgroup
- [Clock] Schedule downtime for all services in this hostgroup
- [Speaker] Enable notifications for all hosts in this hostgroup
- [Speaker] Disable notifications for all hosts in this hostgroup
- [Speaker] Enable notifications for all services in this hostgroup
- [Speaker] Disable notifications for all services in this hostgroup
- [Checkmark] Enable active checks of all services in this hostgroup
- [X] Disable active checks of all services in this hostgroup

For example, if we had twenty web servers that we knew we were going to be down, this would be much easier than scheduling the downtime for each of them individually!

## There's more...

It's worth noting that hostgroups can have subgroups, meaning that all of the hosts added to any of their subgroups are implicitly added to the parent group.

For example, we can define the webservers hostgroup as a subgroup of the servers hostgroup using the `hostgroup_members` directive, as follows:

```
define hostgroup {
    hostgroup_name    servers
    alias             Servers
    hostgroup_members  web-servers
}
```

This would allow us to add hosts to both groups implicitly without needing to refer to the parent group and with all the services assigned to both the groups assigned to the host. Take a look at the following code:

```
define host {
    use                linux-server
    host_name          athens.example.net
    alias              athens
    address            192.0.2.22
    hostgroups         web-servers
}
```

This can be very useful to sort “subcategories” of services. Other examples might include a `dns-servers` group with the `dns-authoritative-servers` and `dns-recursive-servers` subgroups or a `database-servers` group with the `oracle-servers` and `mysql-servers` subgroups.

## See also

- *Creating a new hostgroup, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new servicegroup, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Running a service on all hosts in a hostgroup, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The *Building groups using regular expressions* recipe in this chapter*
- *The *Using inheritance to simplify configuration* recipe in this chapter*



# Building groups using regular expressions

In this recipe, you'll learn a shortcut to building groups of hosts using regular expressions tested against their hostnames.

This recipe is likely only of use to you if you use a naming convention for your hosts that allows them to be reasonably grouped by location, function, or some other useful metric by a shared string in their hostnames.

# Getting ready

You will need to have a server running Nagios Core 4.0 or later, have access to the command line to change its configuration, and understand the basics of how host and servicegroups work. These are covered in the *Creating a new hostgroup* and *Creating a new servicegroup* recipes in [Chapter 1, Understanding Hosts, Services, and Contacts](#).

In this example, we'll group three existing hosts named `web-server-01`, `web-server-02`, and `web-server-03` into a new hostgroup, `web-servers`, based only on their hostnames.

It would help to have some familiarity with regular expressions, but the recipe includes a simple example that should meet many use cases for this trick. An excellent site about regular expressions including tutorials can be found at <http://www.regular-expressions.info/>.



# How to do it...

We can build a hostgroup by matching regular expressions to hostnames like so:

1. Change to the Nagios configuration directory. In the default installation, this is `/usr/local/nagios/etc`. Edit the `nagios.cfg` file in this directory via the following code:

```
# cd /usr/local/nagios/etc
# vi nagios.cfg
```

2. Search for the `use_regexp_matching` directive. Uncomment it if necessary and set it to 1, as follows:

```
use_regexp_matching=1
```

3. Search for the `use_true_regexp_matching` directive. Uncomment it if necessary and ensure that it's set to 0, which it should be by default. Use the following code for this:

```
use_true_regexp_matching=0
```

4. Change to the objects' configuration directory. In the default installation, this is `/usr/local/nagios/etc/objects`. If you have already followed the *Grouping configuration files in directories* recipe in this chapter, your own directory may differ. Take a look at this line:

```
# cd /usr/local/nagios/etc/objects
```

5. In an appropriate file, perhaps `hostgroups.cfg`, add a definition similar to the following. In this case, `.+` means "any string at least one character in length"; you will probably want a pattern of your own devising appropriate to your own host names:

```
define hostgroup {
    hostgroup_name    web-servers
    members           web-server-.+
    alias             Web Servers
}
```

6. Now, validate the configuration and restart the Nagios Core server via the following line:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, if your regular expression was correctly formed to match all the appropriate hostnames, you should find that the hosts become part of the group:

| Web Servers (web-servers) |         |           |                                                                                                                                                                                                                                                       |
|---------------------------|---------|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Host                      | Status  | Services  | Actions                                                                                                                                                                                                                                               |
| web-server-01.example.net | UP      | 2 PENDING |    |
| web-server-02.example.net | PENDING | 2 PENDING |    |
| web-server-03.example.net | PENDING | 2 PENDING |    |

## How it works...

With the `use_regexp_matching` directive set to 1, Nagios Core will attempt to use hostname strings containing the `*`, `?`, `+`, or `\.` strings as regular expressions to match against hostnames. As `web-server-01`, `web-server-02`, and `web-server-03` all match the regular expression `web-server-.*` given in the `members` directive for the `web-servers` hostgroup, all three hosts are added to the group.

We will keep `use_true_regexp_matching` off. If it were on, it would use every hostname pattern as a regular expression, whether or not it had any special regular expression characters. This is probably not what you want for most configurations; for example, if you had a host named `database` that you wanted to add to a group, specifying `database` for the `members` directive would add *all* hosts with a host name matching this string, including, for example, `database-frontend`, `customer-database`.

## There's more...

This matching works in other places aside from hostgroup definitions; for example, you can also use it in the host\_name definitions for services, as follows:

```
define service {  
    use generic-service  
    host_name web-server-.*  
    service_description HTTP  
    check_command check_http  
}
```

This is one of a number of very good suggestions for simplifying object definitions suggested in the Nagios Core manual:

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/objecttricks.html>

## See also

- *Creating a new hostgroup*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- *Creating a new servicegroup*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- The *Building groups using regular expressions* recipe in this chapter
- The *Using inheritance to simplify configuration* recipe in this chapter



# Using inheritance to simplify configuration

In this recipe, you'll learn how to use **inheritance** to handle the situation where hosts and services share a lot of values in common, leading to a large amount of undesirable redundancy in configuration.

Some Nagios objects, particularly hosts and services, have a rather long list of possible directives and the default values for these are not always suitable. It's therefore worthwhile to be able to declare the values you want for these directives once and then spend only a few lines on the actual host definition by copying these values from a template, making the configuration shorter and easier to read.

The previous examples in this book demonstrated the use of this in suggesting you inherit from the `linux-server` host template or the `generic-service` template for the sake of brevity; in this example, we'll define our own templates and show how these can be used to streamline a configuration.

# Getting ready

You will need to have a server running Nagios Core 4.0 or later, have access to the command line to change its configuration, and be familiar with how to define hosts and services. These are covered in the *Creating a new host* and *Creating a new service* recipes in [Chapter 1](#), *Understanding Hosts, Services, and Contacts*.

In this example, we'll define a `critical-host` template, which we'll use as the basis for any host that needs to be checked and notified around the clock with a very stringent `check_command` and with all the notification types enabled. We'll also define two hosts named `phobos.example.net` and `deimos.example.net` that inherit from this template.



# How to do it...

We can define a host template and then define the hosts to inherit from it like so:

1. Change to the objects' configuration directory. In the default installation, this is `/usr/local/nagios/etc/objects`. If you have already followed the *Grouping configuration files in directories* recipe in this chapter, your own directory may differ. Run the following line:

```
# cd /usr/local/nagios/etc/objects
```

2. In an appropriate file, perhaps `templates.cfg`, add the following definition. Note the use of the special directives `name` and `register`:

```
define host {
    name                critical-host
    register            0
    check_command        check_ping!25,10%!50,20%
    max_check_attempts   3
    check_interval       1
    notification_interval 1
    notification_period   24x7
    notification_options d,u,r,f,s
    check_period          24x7
    contact_groups        admins
}
```

3. In another file, or separate files if you prefer to keep your hosts to one per file, define the hosts to inherit from this template. Add in the remaining required directives for hosts and include a `use` directive referring to the established template, as follows:

```
define host {
    use                critical-host
    host_name          phobos.example.net
    alias              phobos
    address            192.0.2.151
}
define host {
    use                critical-host
    host_name          deimos.example.net
    alias              deimos
    address            192.0.2.152
}
```

4. Validate the configuration and restart the Nagios Core server via the following lines:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

You might receive warnings about no services being defined for the hosts, but you can ignore these for now.

With this done, two new hosts should be registered in your configuration: `phobos.example.net` and `deimos.example.net`.

| Host   | Status   | Last Check   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| deimos.example.net                                                                      | UP                                                                                                                                                                         | 10-18-2015 23:12:21                                                                                                                                                            |
| phobos.example.net                                                                      | UP                                                                                                                                                                         | 10-18-2015 23:11:47                                                                                                                                                            |

No other new hosts should be added as the template itself was not explicitly registered with the register directive.

# How it works...

The configuration added in the preceding section defines a new template with the name directive of `critical-host`. As the value for the `host's register` directive is `0` where it normally defaults to `1`, the host is not registered as an object with Nagios Core. Instead, it becomes a snippet of configuration that can be referenced by real objects of the same type by referring to its name.

## Tip

Note that in the template, the normally required values of `host_name`, `alias`, and `address` are missing; this means that the host is not complete and wouldn't work if we tried to register it as an actual host anyway.

Instead, we used its values as the basis for other hosts, `phobos.example.net` and `deimos.example.net`. Both of these hosts inherit from `critical-host` and fill in the rest of the missing values in their own definitions. This freed us from having to repeat the same directives in two different hosts.

If an object inherits a value for a directive from its parent, it's possible to **override** this directive by redefining it in the inheriting object definition. For example, if we wanted `phobos.example.net` to have a different value for `max_check_attempts`, we would only need to add this to its definition. Take a look at the following code:

```
define host {
    use                critical-host
    host_name          deimos.example.net
    alias              deimos
    address            192.0.2.152
    max_check_attempts 5
}
```

## There's more...

The most important thing to note about templates is that they work for a variety of Nagios Core objects, including, most importantly, hosts, services, and contacts. You can, therefore, set up and inherit from a service template in the same way, as follows:

```
define service {
    name          critical-service
    register      0
    ...etc...
}
define service {
    use           critical-service
    ...etc...
}
```

Alternatively, you can use the following code for a contact object:

```
define contact {
    name          critical-contact
    register      0
    ...etc...
}
define contact {
    use           critical-contact
    ...etc...
}
```

Also, potentially useful to note is that inheritance can stack. For example, `critical-host` could itself inherit from a template, perhaps `generic-host`, by adding its own `use` directive, as follows:

```
define host {
    use           generic-host
    name          critical-host
    register      0
    ...etc...
}
```

This allows us to set up an inheritance structure of arbitrary complexity, but you should avoid too much depth to prevent confusing yourself or anyone else who tries to read your configuration. Two levels is probably a sensible limit.

The rules around how inheritance is handled are discussed in more depth in the Nagios Core manual, including a treatment of multiple inheritance. It's useful to know about this, but in the interest of keeping the configuration clear, it's best used sparingly. Refer to the following link:

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/objectinheritance.htm>

## See also

- *Creating a new host, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new service, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *Creating a new contact, [Chapter 1](#), Understanding Hosts, Services, and Contacts*
- *The [Configuring host roles using groups](#) recipe in this chapter*



# Defining macros in a resource file

In this recipe, you'll learn how to define custom user macros in resource files. This is good practice for strings used in `check_command` definitions or other directives that are shared by more than one host or service. For example, take a look at the following example of writing the full path in a `command_name` directive:

```
command_name=/usr/local/nagios/libexec/check_ssh $HOSTADDRESS$
```

In lieu of this, we could write the following line:

```
command_name=$USER1$/check_ssh $HOSTADDRESS$
```

As a result, if the location of the `check_ssh` script changes, we only need to change the value of `$USER1$` in the appropriate resource file to update all of its uses throughout the configuration.

Most of the macros in Nagios Core are defined automatically by the monitoring server, but up to 32 user-defined macros can be used as well in the `$USERn$` form.

# Getting ready

You will need to have a server running Nagios Core 4.0 or later and have access to the command line to change its configuration, in particular the `resource.cfg` file.

In this example, we'll add a new macro, `$USER2$`, to contain the SNMP community name `snagmp` as is used for various `check_snmp` requests.



# How to do it...

We can define our user macro like so:

1. Change to the Nagios configuration directory. In the default installation, this is `/usr/local/nagios/etc`. Edit the `resource.cfg` file in this directory, as follows:

```
# cd /usr/local/nagios/etc
# vi resource.cfg
```

2. Ensure that the `$USER2$` macro is not already defined in the file. If it is, we could define `$USER3$` instead and so on.
3. Add the definition to the end of the file, as follows:

```
$USER2$=snagmp
```

4. Change to the objects configuration directory for Nagios Core. In the default installation, this is `/usr/local/nagios/etc/objects`. Run the following line:

```
# cd /usr/local/nagios/etc/objects
```

5. Edit any of the object configuration files in which you wish to use the value of the macro and replace the inline values with `$USER2$`, as shown in the following code. In our example, we might find various uses of the literal `checksnmp` community string:

```
define command {
    ...
    command_line $USER1$/check_snmp -H $HOSTADDRESS$ -C snagmp -o
    .1.3.6.1.2.1.1.5.0 -r $ARG1$
}
define service {
    ...
    check_command check_snmp!snagmp
}
```

We could swap these out to use the macro instead:

```
define command {
    ...
    command_line $USER1$/check_snmp -H $HOSTADDRESS$ -C $USER2$ -o
    .1.3.6.1.2.1.1.5.0 -r $ARG1$
}
define service {
    ...
    check_command check_snmp!$USER2$
}
```

6. Validate the configuration and restart the Nagios Core server via the following code:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, the monitoring should work the same way as it did before, but we will now use **macro expansion** to centralize the configuration.

# How it works...

Before Nagios Core processes directives such as `command_line` and `check_command`, it will expand all the macros referenced, including the user-defined macros we added in the resource file `resource.cfg`.

One common use for the `$USERn$` macros is for defining the full path of directories where Nagios Core resources such as plugins or event handler scripts are located. In fact, the sample configuration included in Nagios Core defines `$USER1$` in `resource.cfg` as `/usr/local/nagios/libexec`, which is the default location for plugin scripts and binaries.

It's worth noting that you can load more than one resource file by adding more `resource_file` directives in `/usr/local/nagios/etc/nagios.cfg`. For example, to load another file called `resource-extra.cfg`, we could add the second line here:

```
resource_file=/usr/local/nagios/etc/resource.cfg  
resource_file=/usr/local/nagios/etc/resource-extra.cfg
```

## There's more...

There's also a security benefit to using resource files; for sensitive information, we can prevent users other than nagios from reading them by making them readable only to the nagios user, as follows:

```
resource.cfg                                     # cd /usr/local/nagios/etc
                                                  # chown nagios.nagios
                                                  # chmod 0600 resource.cfg
```

This makes it a decent way to store credentials such as usernames and passwords, for example, to be used in check\_command for MySQL, as is shown in the following code:

```
define command {
    command_name    check_mysql_secure
    command_line    check_mysql -H $HOSTADDRESS$ -u naguser -d nagdb -p
$USER3$
}
```

## See also

- *Monitoring the output of an SNMP query, [Chapter 5](#), Monitoring Methods*
- *Monitoring individual nodes in a cluster, [Chapter 8](#), Managing Network Layout*



# Using another object's directives in a host or service check

In some cases where checks of hosts or services depend on each other, we may need to use information in a definition for `check_command` that's dependent on a property of another host or service. In these cases, you may end up with an unnecessary duplication of this information, so if it changes in one place, the check may break if it's not changed in another. These situations can be hard to remember and can cause confusing errors when forgotten; so, ideally, we would want our configuration to be more dynamic so that we only have to change such pieces of information in one place.

For example, to check whether two BGP routing peers are correctly established and that they exchange routes via SNMP, we would need the value of address of the other one so that for each of them, we know which peer to check. Suppose we have these two hosts configured based on a `bgp-router` template; in this case, we would execute the following code:

```
define host {
    use          bgp-router
    host_name    bgp-router-01.example.net
    alias        bgp-router-01
    address      192.0.2.96
}
define host {
    use          bgp-router
    host_name    bgp-router-02.example.net
    alias        bgp-router-02
    address      192.0.2.97
}
```

If we wanted to apply a `check_bgp_peer` command to each of them to verify that the BGP data retrieved from SNMP for both of them is valid, we would need to refer to the address of the other host in `check_command`; it may look something similar to this:

```
define service {
    use          snmp-service
    host_name    bgp-router-01.example.net
    service_description BGP_PEER_ROUTER_02
    check_command check_bgp_peer!192.0.2.97
}
define service {
    use          snmp-service
    host_name    bgp-router-02.example.net
    service_description BGP_PEER_ROUTER_01
    check_command check_bgp_peer!192.0.2.96
}
```

This would work, but it means that if the IP address of either of the peers were to change, we'd need to change it in two places; in the address of the host object and the `check_command` of its peer's service check. In this recipe, we'll arrange for the

configuration to refer to the address of the other host rather than repeating it in the configuration using a special syntax of macros. These are known in Nagios as **on-demand macros**.

# Getting ready

You will need to have a server running Nagios Core 4.0 or later, access to the command line to change its configuration, and a basic understanding of how directives apply to hosts and services and how hosts and services refer to one another. These are discussed in chapters 1 and 2.

You will also need at least two hosts configured that are in a situation where it's necessary for at least one of their services to refer to the configured values of the other host, as with our example.



# How to do it...

We can create a reference to the values of another host's directives like so:

1. Change to the objects' configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your hosts and services in a different file, move to its directory instead via the following line of code:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the service that contains the attribute you want dynamically expanded—that is, the `check_command` that uses the value of the directive of another host. In this case, we want to edit the `check_command` of our `BGP_PEER_ROUTER_01` and `BGP_ROUTER_01` services and replace the hardcoded IP address of the peer in the lines highlighted here:

```
define service {
    use                snmp-service
    host_name          bgp-router-01.example.net
    service_description BGP_PEER_ROUTER_02
    check_command       check_bgp_peer!192.0.2.97
}
define service {
    use                snmp-service
    host_name          bgp-router-02.example.net
    service_description BGP_PEER_ROUTER_01
    check_command       check_bgp_peer!192.0.2.96
}
```

3. Replace the data you want dynamically expanded with a macro in the following format:

**`$HOST<macro>:<hostname>$`**

In this case, we want the `$HOSTADDRESS$` macro of the `bgp-router-02` host in the first service and the same property for the `bgp-router-01` host in the second, so our new configuration will look similar to this:

```
define service {
    use                snmp-service
    host_name          bgp-router-01.example.net
    service_description BGP_PEER_ROUTER_02
    check_command       check_bgp_peer!$HOSTADDRESS:bgp-
router02.example.net$
}
define service {
    use                snmp-service
    host_name          bgp-router-02.example.net
    service_description BGP_PEER_ROUTER_01
    check_command       check_bgp_peer!$HOSTADDRESS:bgp-
router01.example.net$
}
```

4. Validate the configuration and restart the Nagios Core server, as follows:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

With this done, the checks we have already established should continue working as normal but with less duplication and a little more clarity in the configuration.

## How it works...

When the on-demand `$HOSTADDRESS:bgp-router-02.example.net$` and `$HOSTADDRESS:bgp-router-01.example.net$` macros are expanded for the execution of the `check_command`, Nagios Core refers to the address directive for a host, just as it normally would with a `$HOSTADDRESS$` macro without the hostname specified. The difference is that with the colon added and the name of another host given, Nagios Core will refer to the value of the macro for this host and not the host for which the service is defined.

In our example, `$HOSTADDRESS:bgp-router-02.example.net$` would expand to `192.0.2.97`, the value of the address directive for `bgp-router-02.example.net`, and `$HOSTADDRESS:bgp-router-01.example.net$` to `192.0.2.96`, the value of the address directive for `bgp-router-01.example.net`. The checks would therefore end up running with the same arguments as they had when they were hardcoded, but the information will not be duplicated in the configuration. It also becomes easier to understand what is checked when the configuration is read by someone else as the hostname for the address is explicitly given.

## There's more...

The `$HOSTADDRESS$` macro is not the only one that can be expanded in this way. The `$HOSTNAME$` and `$HOSTALIAS$` macros and even dynamic information such as `$HOSTSTATE$` can be expanded as an on-demand macro for another host.

Similarly, hosts are not the only object for which this works. You can expand macros for contacts, hostgroups, and servicegroups in the same way. For example, `$CONTACTADDRESS1:admin$`, `$HOSTGROUPALIAS:webserver$`, and `$SERVICEGROUPNOTES:snmp-checks$` would all work if given objects of the appropriate type with these names.

You can refer to service objects with a similar syntax, allowing you to specify both the host name and the service description for the service that has the values you want to reference by adding another colon and `service_description` for the service; for example, `$SERVICESTATE:webserver-01:HTTP$` would work for a service check called HTTP on a host named `webserver-01`.

The Nagios Core documentation goes into more depth about the possibilities for on-demand macros at

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/macros.html>.

## See also

- *Customizing an existing command*, [Chapter 2](#), *Working with Commands and Plugins*
- The *Using custom directives* recipe in this chapter
- The *Using inheritance to simplify configuration* recipe in this chapter
- The *Dynamically building host definitions* recipe in this chapter



# Using custom directives

While directives for hosts such as `host_name`, `alias`, and `address` may be adequate for many purposes, depending on our monitoring requirements, we may sometimes need Nagios Core to be able to refer to other data that's specific to individual hosts but can't be included in one of these directives.

For example, it's common in networking to have a **management interface** for a device. This is a network interface that links to a network used only by the administrators of the device for the purposes of monitoring and configuring it and is not accessible to users of the service. While we may want to run, for example, `check_ping` against the public-facing address of a given host, we may also need to verify that a service at a different address on a completely different network is running.

We might have a host configured in the following way referring to a webserver and including a PING check to make sure its public-facing interface with the `203.0.113.1` address is up and running:

```
define host {
    use                webservice
    host_name          webservice-01.example.net
    alias              webservice-01
    address            203.0.113.1
}
define service {
    use                generic-service
    host_name          webservice-01
    service_description PING
    check_command      check_ping_addr!$HOSTADDRESS$
}
```

Suppose this host also had a second network address for management purposes only, `192.0.2.1`, and we also wanted to arrange to send PING requests to this address to ensure it responded. We could set up another service `PING_MGMT` check like so, by hardcoding the address into the value of the `check_command` directive:

```
define service {
    use                generic-service
    host_name          webservice-01
    service_description PING_MGMT
    check_command      check_ping_addr!192.0.2.1
}
```

However, if we have to check multiple services this way, we may find that we're repeating ourselves in specifying the management address. It would be helpful to add our own property for the management address to the host object so that we could refer to it in services. We'll arrange this in this recipe.

# Getting ready

You will need to have a server running Nagios Core 4.0 or later, access to the command line to change its configuration, and a basic understanding of how directives apply to hosts and services and how hosts and services refer to one another. These are discussed in chapters 1 and 2.



# How to do it...

We can create a reference to the values of another host's directives like so:

1. Change to the objects configuration directory for Nagios Core. The default is `/usr/local/nagios/etc/objects`. If you've put the definition for your hosts and services in a different file, move to its directory instead. Run the following line:

```
# cd /usr/local/nagios/etc/objects
```

2. Edit the file containing the definition for the host for which you want the custom attribute. In this example, we will add it to our `webserver-01` host declaration, as follows:

```
define host {
    use          webservers
    host_name    webserver-01.example.net
    alias        webserver-01
    address      203.0.113.1
}
```

3. Add the directive with the name needed and its corresponding value along with an underscore prefix (`_`), as follows:

```
define host {
    use          webservers
    host_name    webserver-01.example.net
    alias        webserver-01
    address      203.0.113.1
    _address_mgmt 192.0.2.1
}
```

4. If needed, configure an appropriate service that uses this value; capitalize the name of the directive and prepend `_HOST` to get the full name of the necessary macro via the following code:

```
define service {
    use          generic-service
    host_name    webserver-01
    service_description PING_MGMT
    check_command check_ping_addr!$_HOSTADDRESS_MGMT$
}
```

5. Validate the configuration and restart the Nagios Core server, as follows:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
```

With this done, we will be able to refer to the custom directive in macros; including specifying new commands or services to use its values.

## How it works...

Nagios Core understands directives prefixed with underscores as **custom object variables**; it allows any alphanumeric string with underscores to be used in defining the directive. The value is made all uppercase and then becomes available for macro expansion in `check_command` definitions in the same way as all standard Nagios Core macros.

## There's more...

The custom object variables feature was established in Nagios Core in response to requests from users to allow adding custom information to their objects and get greater flexibility out of the commands that use the information stored in them. This means it works in other object types too, not just hosts; other possibilities for uses of custom declarations might be as follows:

- MAC addresses for hosts, for example, `$_HOST_MACADDRESS$`
- Latitude, longitude, and elevation for hosts, for example, `$_HOST_LATITUDE$`
- Email, instant message, or PGP key IDs for contact objects, for example, `$_CONTACT_PGP$`

### Note

The Nagios Core documentation includes some more detail on the usage of custom macros at

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/customobjectvars.ht>

## See also

- *Customizing an existing command*, [Chapter 2](#), *Working with Commands and Plugins*
- The *Using another object's directives in a host or service check* recipe in this chapter
- The *Using inheritance to simplify configuration* recipe in this chapter
- The *Dynamically building host definitions* recipe in this chapter



# Dynamically building host definitions

In this recipe, you'll learn one possible method of building Nagios configuration dynamically to avoid having to compose or copy and paste a lot of directives for new hosts or services. In other words, this recipe is about generating a configuration using templates.

To demonstrate how this is useful, we'll use the m4 macro language utility, which should be available on virtually any UNIX-like system, including GNU/Linux and BSD. As a tool designed for macro expansion, m4 is particularly well suited to creating verbose plain text configuration files, such as the ones used by Nagios Core.

The principles here should apply just as easily to your favored programming or templating language, perhaps Python, Perl, or shell scripts.

# Getting ready

You will need to have the `m4(1)` macro language tool available to you, preferably but not necessarily on the same system as the one running Nagios Core. It is a very standard tool and should be already installed or available as part of a package. The version used in this example is GNU `m4(1)`, documented at <http://www.gnu.org/software/m4/manual/m4.html>. This recipe does not assume any familiarity with `m4` and will show you the basics.

You may like to work in a new subdirectory in your home directory; run the following code for this:

```
$ mkdir $HOME/nagios-dynamic-hosts
$ cd $HOME/nagios-dynamic-hosts
```

# How to do it...

We can create and apply an example Nagios Core configuration template like so:

1. Create a new file `host-service-template.m4` with the following contents:

```
define(`NAGHOST', `
    define host {
        host_name            $1
        alias                 $2
        address               $3
        contact_groups        ifelse(`$#', `4', `$4', `admins')
        check_command          check-host-alive
        check_interval         5
        check_period           24x7
        max_check_attempts     3
        notification_interval  30
        notification_options    d,r
        notification_period    24x7
    }
    define service {
        host_name            $1
        contact_groups        ifelse(`$#', `4', `$4', `admins')
        check_command          check_ping!100,10%!200,20%
        check_interval         5
        check_period           24x7
        max_check_attempts     3
        notification_interval  30
        notification_period    24x7
        retry_interval         1
        service_description    PING
    }
`)
```

2. Create a second file in the same directory called `sparta.host.m4` with the following contents:

```
include(`host-service-template.m4')
NAGHOST(`sparta', `Sparta Webserver', `192.0.2.21')
```

3. Create a third file in the same directory called `athens.host.m4` with the following contents:

```
include(`host-service-template.m4')
NAGHOST(`athens', `Athens Webserver', `192.0.2.22', `ops')
```

4. Run the following commands and note the output:

```
$ m4 sparta.host.m4
define host {
    host_name            sparta
    alias                 Sparta Webserver
    address               192.0.2.21
    contact_groups        admins
    check_command          check-host-alive
    check_interval         5
```



```

        check_period          24x7
        max_check_attempts    3
        notification_interval  30
        notification_options   d,r
        notification_period    24x7
    }
    define service {
        host_name              sparta
        contact_groups          admins
        check_command            check_ping!100,10%!200,20%
        check_interval          5
        check_period             24x7
        max_check_attempts       3
        notification_interval    30
        notification_period      24x7
        retry_interval           1
        service_description      PING
    }
$ m4 athens.host.m4
define host {
    host_name              athens
    alias                  Athens Webserver
    address                192.0.2.22
    contact_groups          ops
    check_command            check-host-alive
    check_interval          5
    check_period             24x7
    max_check_attempts       3
    notification_interval    30
    notification_options     d,r
    notification_period      24x7
}
define service {
    host_name              athens
    contact_groups          ops
    check_command            check_ping!100,10%!200,20%
    check_interval          5
    check_period             24x7
    max_check_attempts       3
    notification_interval    30
    notification_period      24x7
    retry_interval           1
    service_description      PING
}

```

As seen in the preceding output, we can now generate a basic host and service configuration with a two-line m4 script referring to a template simply by writing the output to a .cfg file. Take a look at this line:

```
$ m4 sparta.host.m4 > sparta.cfg
```

## How it works...

The `sparta.host.m4` and `athens.host.m4` files both called an `m4` macro with arguments after including the template for the host and service in the `host-service-template.m4` file. This was expanded into the full definition and the arguments given were substituted as follows:

- `$1` was replaced with the first argument, `host_name`
- `$2` was replaced with the second argument, `alias`
- `$3` was replaced with the third argument, `address`
- `$4` was replaced with the fourth argument, `contact_group`

### Tip

Note that two of these values, `$1` and `$4`, were used in both the host and the `PING` service definitions.

Also, note that the `$4` argument is optional; the `ifelse` construct tests the number of arguments and, if it finds that there are four, it uses the value of the fourth argument. For `athens.example.net`, this is the contact group `ops`. If there is no fourth argument, it defaults instead to the `admins` value. This allows us to set **default** values for arguments if we so choose.

The rest of the directives are all written directly into the template. The configuration made by this process is valid for Nagios Core, assuming the `check_command` and `contact_groups` used are defined.

## There's more...

To automate things even further, we could use `make(1)` to automatically generate `.cfg` files from anything with the `.host.m4` extension with the following Makefile syntax:

```
%.cfg : %.host.m4
    m4 $< > $*.cfg
```

### Tip

Note that correct Makefile syntax usually requires a literal *Tab* character to indent the second line, not four spaces.

With this in the same directory as all the preceding files, in order to build the configuration for the `sparta.example.net` host, we would only need a call to `make(1)` to generate the file, as follows:

```
$ make sparta.cfg
m4 sparta.host.m4 > sparta.cfg
$ make athens.cfg
m4 athens.host.m4 > athens.cfg
```

### Tip

Note that it's best practice to avoid repeating directives and use hostgroups and host and service templates instead to define “roles” for new hosts. This makes adding and removing the hosts much easier and both the processes are explained in this chapter in the *Configuring host roles using groups* and *Using inheritance to simplify configuration* recipes. Mei Douthitt goes into considerably more depth about the possibilities of using m4 for Nagios configuration at

<http://administratosphere.wordpress.com/2009/02/19/configuring-nagios-with-m4/>.

## See also

- The *Configuring host roles using groups* recipe in this chapter
- The *Using inheritance to simplify configuration* recipe in this chapter



# Chapter 10. Security and Performance

In this chapter, we will cover the following recipes:

- Using authentication for the Nagios Core web interface
- Using authenticated contacts
- Writing debugging information to the Nagios log file
- Monitoring Nagios performance with nagiosstats
- Setting up a redundant monitoring host

# Introduction

Most administrators of midsize networks choose to dedicate an entire server to monitor software and, sometimes, a whole server just for Nagios Core. This is because of the following two main factors that are common to most comprehensive Nagios Core setups:

- **They have a lot of privileges** because in order to inspect the running state of so many different hosts and services, they need to be conferred with an appropriate network to those services. This often means that their IP addresses are whitelisted all over the network. A user who is able to assume this privilege could potentially do a lot of damage.
- **They have a lot of work to do** and hence ideally have dedicated software and hardware resources to run what can be thousands of host and service checks smoothly and to promptly notice problems and recoveries. If a Nagios Core server is not able to keep up with its check schedule, it could cause delays in notification about very important services.

It's, therefore, very important to take into account the security and performance of your Nagios Core server when building your configuration.

The general best network security practices apply to Nagios Core management and will not be discussed in depth here. The following general guidelines are just as relevant to securing a Nagios Core server as they are to any other kind of server:

- **Don't run the server as root:** This is probably not a concern unless you've changed it yourself, for example, if you've installed Nagios using the Quick Start Guides, the server should be set up to run as the unprivileged nagios user. Nagios Core should not need root privileges under most circumstances. If you do need to provide a process with the privileges of another user, you should consider using `sudo(8)`.
- **Use a firewall:** The protections offered by Nagios Core, NSCA, and NRPE's host checking are very basic and are not a replacement for a software or hardware firewall policy. Even a simple `iptables` or `pf` software firewall would be a very good idea to protect a monitoring server.
- **Use the principle of least privilege:** Don't confer Nagios Core or any of its plugins or processes more privileges than they need and lock down writable files, such as logs and state information, only to appropriate users. Similarly, only allow Nagios Core access through firewalls for what it needs to check and nothing else.
- **Encrypt sensitive information:** Don't put credentials in plain text in any of the configuration files if you can avoid it and, if you can't avoid it, define them in resource files that are only readable by the nagios user.

You should therefore not consider this chapter a complete guide to securing and optimizing your Nagios Core server. For a more comprehensive treatment of security and optimization procedures, be sure to take a look at these pages in the Nagios Core documentation:

- Security Considerations:

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/security.html>

- Enhanced CGI Security and Authentication:

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/cgisecurity.htm>

- Tuning Nagios for Maximum Performance:

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/tuning.html>

In particular, this chapter focuses on securing the CGI scripts, as the web interface readily exposes Nagios Core's information to the world and is particularly vulnerable to abuse if misconfigured. It will also include methods to assess how Nagios Core performs.





# Using authentication for the Nagios Core web interface

In this recipe, we'll explore the use of basic authentication for the Nagios Core web interface, probably the single most important configuration step in preventing abuse of the software by malicious users.

By default, the Nagios Core installation process takes the sensible step of locking down the CGI scripts in its recommended Apache configuration file with standard HTTP authentication for a default user named `nagiosadmin`, with full privileges.

Unfortunately, some administrators take the step of removing this authentication or never installing it despite the recommendations in the installation guide. It's a good idea to install it and keep it in place even on private networks, especially if the server running Nagios Core is open to the Internet in any way (generally not advised).

Keeping authentication on is useful not just because of the security benefits, but also because it allows you to set up basic access control, allowing certain users permission to read state or run commands on certain resources, but not on others. It also has other more subtle benefits, such as recording the names of users that carry out actions for logging purposes.

# Getting ready

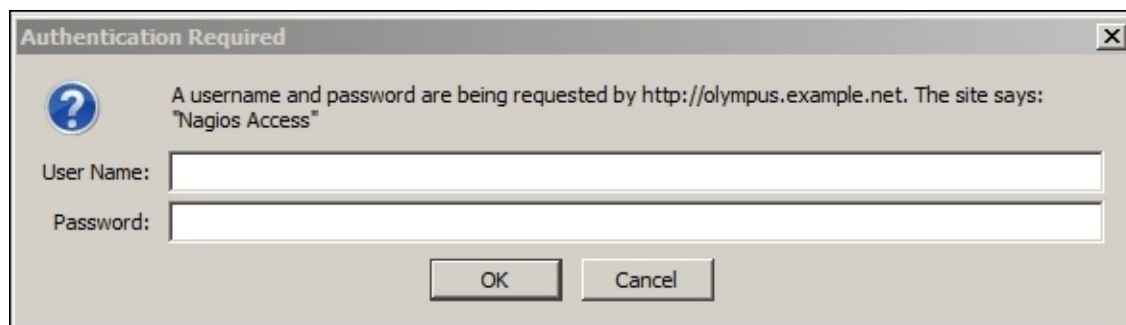
You'll need access to the backend of a Nagios Core server with version 4.0 or greater to change its configuration and restart it. You'll also need a functioning web interface. Much of this recipe will assume that the web interface is running on the recommended Apache HTTPD server; you may also need to be able to edit this configuration. Some familiarity with Apache HTTPD is assumed here; the documentation available online regarding Apache is excellent if you need to consult it (<https://httpd.apache.org/docs/>).

This recipe will be covered in two parts: we'll first ensure that all the recommended settings are in place to properly require authentication and we'll then demonstrate how to add a user `helpdesk` with read-only permissions for the server's web interface; they will be able to read the states of all the hosts and services, but the user will not be able to (for example) issue commands or submit passive check results.

# How to do it...

We can ensure that proper authentication is in place for the Nagios Core web interface by following these steps:

1. Clear your browser of all cookies and saved authentication data and try to visit the web interface. If your browser does not challenge you for a username or password, as shown in the following screenshot, it's likely that your authentication is disabled or not working correctly:



2. On the server, move to the Nagios Core configuration directory and open the `cgi.cfg` file. In the default installation, this is saved in `/usr/local/nagios/etc/cgi.cfg`:

```
# cd /usr/local/nagios/etc
# vi cgi.cfg
```

3. Ensure that the value for `use_authentication` is uncommented and set to 1:

```
use_authentication=1
```

4. Ensure that the `default_user_name` directive is commented out:

```
#default_user_name=guest
```

5. In the `nagios.conf` file in your Apache configuration, verify that the following lines are included to refer to the `htpasswd.users` file:

```
AuthName "Nagios Access"
AuthType Basic
AuthUserFile /usr/local/nagios/etc/htpasswd.users
Require valid-user
```

This file should be somewhere in the configuration directory for Apache, for example, `/etc/apache2/conf-available/nagios.conf`. If you need to make changes to Apache's configuration to fix this, you will need to restart Apache to have the changes take effect.

We can add a read-only user named `helpdesk` as given in the next step.

6. Add the user to the `htpasswd.users` file using the `htpasswd` command and the Apache HTTPD password manager. Its location will vary depending on your system; a few common locations for it are `/usr/bin` and `/usr/local/apache/bin`:

```
# htpasswd /usr/local/nagios/etc/htpasswd.users helpdesk
New password:
Re-type new password:
Adding password for user helpdesk
```

You may like to make the `htpasswd.users` file only readable using the web server user if you are concerned about the hashes being stolen by users on the system.

7. In `cgi.cfg`, uncomment the `authorized_for_read_only` directive and add the new helpdesk user to its value:

```
authorized_for_read_only=helpdesk
```

8. Add the user to the values of the `authorized_for_all_services` and `authorized_for_all_hosts` directives:

```
authorized_for_all_services=nagiosadmin,helpdesk
authorized_for_all_hosts=nagiosadmin,helpdesk
```

Be careful not to confuse these with the `authorized_for_all_service_commands` and `authorized_for_all_host_commands` directives.

You don't need to restart the Nagios Core server for these changes to take effect as you normally would with other `.cfg` files.

With this done, you should only be able to access the Nagios Core web interface with a valid username and password. The default `nagiosadmin` user created on installation should have full, and the `helpdesk` user added in this recipe should be able to view host and service states, but it will be unable to issue any commands, such as rescheduling checks or submitting passive check results.

## How it works...

It's important to note that it isn't actually Nagios Core itself that is prompting for the username and password and running the authentication check; this is a function performed by the web server, as specified in the `nagios.conf` file recommended for installation with Apache HTTPD.

After login, however, Nagios Core uses the permissions defined in the `cgi.cfg` file each time one of the CGI scripts in the web interface is accessed. It does this to ensure that the user, as authenticated by the web server, has permission to view the requested page or execute the requested action.

We disable the `default_user_name` directive by commenting it out because this specifies a user that Nagios Core will *assume* for users who access the CGIs without authentication. This is a potentially dangerous setting and is best avoided in most circumstances, particularly with a server that has publicly accessible addresses.

The following directives in `cgi.cfg` allow us to refine permissions in order to use the CGIs to form a simple kind of access control list:

- `authorized_for_configuration_information`: In this directive, users are allowed to view configuration information for hosts in the web interface
- `authorized_for_system_information`: In this directive, users are allowed to view Nagios's process and performance information
- `authorized_for_system_commands`: In this directive, users are allowed to run commands that affect the Nagios process, such as shutdowns and restarts
- `authorized_for_all_services`: In this directive, users are allowed to view status information and history for all services
- `authorized_for_all_hosts`: In this directive, users are allowed to view status information and history for all hosts
- `authorized_for_all_service_commands`: In this directive, users are allowed to run commands on all services, such as rescheduling checks or submitting passive check results
- `authorized_for_all_host_commands`: In this directive, users are allowed to run commands on all hosts, such as rescheduling checks or submitting passive check results
- `authorized_for_read_only`: In this directive, users with read-only rights to the CGIs cannot run commands or make comments

Furthermore, the process of refining access per service and per host is done with authenticated contacts, which is demonstrated in the *Using authenticated contacts* recipe in this chapter. This is highly recommended for teams with mixed responsibilities that require access to the same Nagios Core web interface.

## There's more...

Aside from authentication via Apache HTTPD, it's also often sensible to limit the IP addresses that are allowed access to the instance using `Order` and `Allow` Apache directives. We could extend the `nagios.conf` file loaded by Apache using this code snippet:

```
<Directory "/usr/local/nagios/sbin">
    Options ExecCGI
    AllowOverride None
    AuthName "Nagios Access"
    AuthType Basic
    AuthUserFile /usr/local/nagios/etc/htpasswd.users
    <RequireAll>
        Require valid-user
        Require ip 192.0.2.1
    </RequireAll>
</Directory>
```

This would only allow `192.0.2.1` to access the CGIs and the user will still have to provide a username and password. The configuration will deny access with `403 Forbidden` to anyone else. Similarly, we could arrange to only allow connections to HTTPS, perhaps with an `SSLRequireSSL` directive; in general, we can configure Apache to carefully control the access to CGIs, let alone abusing them.

### Tip

Note that none of this should take the place of an appropriate firewall solution and policy. A monitoring server should be protected as carefully as any other mission-critical server.

## See also

- The *Using authenticated contacts* recipe in this chapter
- *Viewing configuration in the web interface*, [Chapter 7](#), *Using the Web Interface*
- *Scheduling checks from the web interface*, [Chapter 7](#), *Using the Web Interface*





# Using authenticated contacts

In this recipe, you'll learn how to use authenticated contacts to refine our control of access to information in the Nagios Core web interface. This recipe is useful in situations where a particular user requires information on the status of certain hosts and services but should not be allowed to view others. This is a setup that can't be managed with the directives in `cgi.cfg`.

As a simple example, on a given monitoring server, we might have two hosts configured as follows:

```
define host {
    use          linux-server
    host_name    sparta.example.net
    alias        sparta
    address      192.0.2.21
    contacts     nagiosadmin
}
define host {
    use          linux-server
    host_name    athens.example.net
    alias        athens
    address      192.0.2.22
    contacts     nagiosadmin
}
```

We might like to add a new user `athensadmin` with permissions to view the status of and run commands on the `athens.example.net` host and its services but not the `sparta.example.net` host or its services.

## Getting ready

You'll need access to the backend of a Nagios Core server with version 4.0 or greater to change its configuration and restart it. You'll also need a functioning web interface with authentication running and be familiar with how it works; the *Requiring authentication for the web interface* recipe in this chapter explains how to do this.

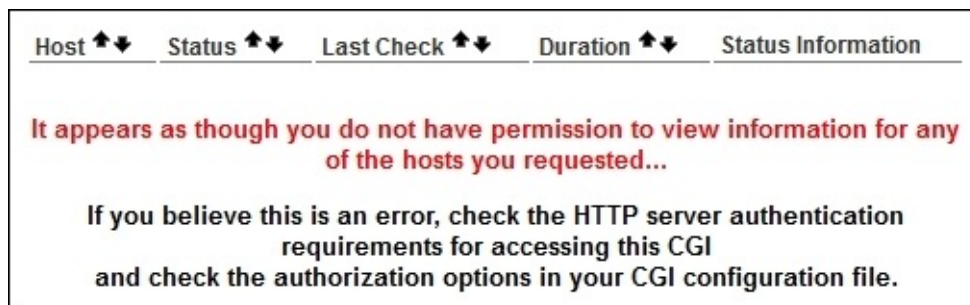
# How to do it...

We can add an authenticated contact as follows:

1. Create a new user in the `htpasswd.users` file named `athensadmin` using the `htpasswd` tool:

```
# htpasswd /usr/local/nagios/etc/htpasswd.users athensadmin
New password:
Re-type new password:
Adding password for user athensadmin
```

2. Log in to the Nagios Core web interface with the credentials you just added and click on **Hosts** on the left-hand side menu to verify that you are not able to view any information yet:



3. Back on the command line, change to the Nagios Core objects configuration directory. In the quickstart guide installation, the directory is `/usr/local/nagios/etc/objects`:

```
# cd /usr/local/nagios/etc/objects
```

4. Edit the `contacts.cfg` file to include a definition of a new contact object, `athensadmin`. Here, we've used the `generic-contact` template. You can use your own template or values if you prefer; note that the value for `contact_name` is the important part here:

```
define contact {
    use                generic-contact
    contact_name       athensadmin
    alias              Athens Administrator
    email              athens@example.com
}
```

5. Edit the hosts or services for which you want to allow the `athensadmin` user access and add `athensadmin` to their lists of contacts. For our example, this means that the definition for `athens.example.net` looks like this:

```
define host {
    use                linux-server
    host_name          athens.example.net
    alias              athens
    address             192.0.2.22
```

```
    contacts    nagiosadmin,athensadmin  
}
```

6. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios restart
```

With this done, logging in as the athensadmin user should allow you to view the details of the athens.example.net host but nothing else.

| Host                | Status  | Last Check  | Duration  | Status Information                        |
|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|-------------------------------------------|
| athens.example.net  | UP                                                                                       | 11-02-2015 21:36:58                                                                          | 0d 0h 0m 22s                                                                               | PING OK - Packet loss = 0%, RTA = 0.22 ms |

You should also be able to issue commands related to that host, such as rescheduling checks and acknowledging issues.

## How it works...

When logging in as an authenticated user with Apache HTTPD, Nagios Core checks whether the username matches the `contact_name` value of any configured contacts. If it does, then the privileges to inspect the state of that contact's associated hosts and services is granted along with the ability to run commands on only those hosts and services. The web interface otherwise works in just the same way. To the authenticated contact, it appears as if no other hosts or services are being monitored.

If your network includes colocated equipment or teams with mixed monitoring responsibilities, this will allow you to restrict the Nagios Core interface to certain hosts for certain users. This can be very useful for confidentiality, transparency, and delegation purposes.

## There's more...

If you want to allow an authenticated user read-only access to the details for their associated hosts or services, you can arrange that by adding their username to the values of the `authorized_for_read_only` directive in `cgi.cfg`:

```
authorized_for_read_only=athensadmin
```

The `athensadmin` user will then still be able to view the same host and service information but will not be able to issue any commands.

### Host Commands

Your account does not have  
permissions to execute  
commands.

## See also

- *Creating a new contact*, [Chapter 1](#), *Understanding Hosts, Services, and Contacts*
- The *Requiring authentication for the web interface* recipe in this chapter





# Writing debugging information to the Nagios log file

In this recipe, you'll learn how to use the debugging log file in Nagios Core to get various kinds of process information from the running program, considerably more than available in the standard `log_file`. This is useful not just for debugging purposes when Nagios Core is doing something unexpected at runtime, but it is also useful to get a better idea of how the server is working in general and with your particular configuration.

# Getting ready

You will need a Nagios Core server with version 4.0 or greater. You will need access to change the `nagios.cfg` file and to restart the server.

In this example, we'll simply log everything we possibly can and then explain how to refine the logging behavior if necessary in the *How it works* section.

# How to do it...

We can enable very verbose debugging for our monitoring server as follows:

1. Change to the configuration directory for Nagios Core. In the default installation, the location is `/usr/local/nagios/etc`. Edit the `nagios.cfg` file:

```
# cd /usr/local/nagios/etc
# vi nagios.cfg
```

2. Look for the `debug_level`, `debug_verbosity`, and `debug_file` directives. Ensure that they are uncommented or add them to the end of the file if they don't exist, and then define them as follows:

```
debug_level=-1
debug_verbosity=2
debug_file=/usr/local/nagios/var/nagios.debug
```

3. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

4. With this done, the `/usr/local/nagios/var/nagios.debug` file should start filling up with information about the running process quite quickly. You may find it instructive to watch it for a while with `tail -f`, which will show you the contents of the file as it updates:

```
# tail -f /usr/local/nagios/var/nagios.debug
[1446454473.296558] [016.1] [pid=2023] ** Async check result for host
'sparta.example.net' handled: new state=1
[1446454473.296565] [064.1] [pid=2023] Making callbacks (type 7)...
[1446454473.296571] [064.2] [pid=2023] Callback #1 (type 7) return code
= 0
[1446454473.296578] [4096.2] [pid=2023] ## 1 descriptors had input
[1446454473.296584] [12288.1] [pid=2023] ## Polling 1500ms; sockets=6;
events=13; iobs=0x2243140
[1446454474.798162] [4096.2] [pid=2023] ## 0 descriptors had input
[1446454474.798204] [064.1] [pid=2023] Making callbacks (type 11)...
[1446454474.798213] [12288.1] [pid=2023] ## Polling 1500ms; sockets=6;
events=13; iobs=0x2243140
```

# How it works...

The `debug_level` directive specifies how much information (and of what kind) should be written to the debugging log. Here, we've used the value `-1`, which is a shortcut for specifying that all debugging information should be written to the debugging log file.

In practice, however, we often only want to get information about particular kinds of Nagios Core tasks. In this case, we can use OR values for `debug_level` to specify which ones.

The different kinds of debugging information can be specified with the following numbers:

- 1: This refers to function enter and exit debugging
- 2: This refers to config debugging
- 4: This refers to process debugging
- 8: This refers to scheduled event debugging
- 16: This refers to host and service check debugging
- 32: This refers to notification debugging
- 64: This refers to event broker debugging

In version 3.3 and later of Nagios Core, the following can also be specified:

- 128: This refers to external commands debugging
- 256: This refers to commands debugging
- 512: this refers to scheduled downtime debugging
- 1024: This refers to comments debugging
- 2048: This refers to macros debugging

Rather than using comma-separated values to specify more than one value, they are added together. For example, if we want to save the process and scheduled event debugging information and nothing else, we would use  $4 + 8 = 12$ :

```
debug_level=12
```

We can turn the debugging off completely by changing `debug_level` back to `0`, its default value.

## There's more...

Nagios Core generates a great deal of information at the highest level of debugging, with over 30 lines per second even on minimal configurations, so be careful not to leave this running permanently if you don't always need it as it can slowly fill a disk. You can avoid this situation using the `max_debug_file_size` directive to specify a maximum size in bytes for the file. For example, to restrict the file to one megabyte, we could set the directive as follows:

```
max_debug_file_size=1000000
```

Nagios Core will *roll* an existing debugging log by adding a `.old` extension when it exceeds this size and will start a new one. It will also automatically delete any previous `.old` logs when it does this.

## See also

- *Viewing and interpreting notification history*, [Chapter 7](#), *Using the Web Interface*
- The *Monitoring Nagios performance with Nagiosstats* recipe in this chapter





# Monitoring Nagios performance with `nagiosstats`

In this recipe, you'll learn how to use the `nagiosstats` utility to get some statistics about the performance of a Nagios Core process and the states of the hosts and services that it monitors.

Optionally, we'll also show you how to use the `mrtg.cfg` file built in the Nagios Core source distribution at the `./configure` time to set up graphs built by `mrtg` (**Multi-Router Traffic Grapher**) and to link to the graphs in the menu of the web interface. The Nagios Core source distribution includes some ready-made files to assist with this, which we'll use here.

# Getting ready

You will need a Nagios Core 4.0 server or newer server installed and running to invoke `nagiosstats`. Older versions do include the utility, but there is not quite as much information that is returned.

If you would like to run the `mrtg` graphing as well, which is highly recommended, you should have `mrtg` and its helper program `indexmaker` installed on your system. If you are already graphing other things with `mrtg`, don't worry, this recipe should not interfere with that.

The recipe does not assume any familiarity with `mrtg`, but if you have any problems with it, you can consult its online documentation at <http://oss.oetiker.ch/mrtg/doc/index.en.html>.

You should also have access to the sources from which your installation of Nagios Core was compiled. If you need to retrieve the sources again, you can download them again from Nagios Core's website, <https://www.nagios.org/>.

In this case, you will need to run `./configure` again to generate the file required, `sample-config/mrtg.cfg`.

# How to do it...

We can invoke `nagiosstats` itself in one step whenever we want to get some statistics about the server's performance, with the following command:

1. Run the following command:

```
# /usr/local/nagios/bin/nagiosstats -c /usr/local/nagios/etc/nagios.cfg
```

This should give you an output beginning with the following:

```
Nagios Stats 4.1.1
Copyright (c) 2003-2008 Ethan Galstad (www.nagios.org)
Last Modified: 08-19-2015
License: GPL
```

## CURRENT STATUS DATA

```
-----
Status File:
/usr/local/nagios/var/status.dat
Status File Age:                0d 0h 0m 8s
Status File Version:           4.1.1

Program Running Time:          0d 0h 4m 27s
Nagios PID:                    2023
...
```

2. Run the following command:

```
# /usr/local/nagios/bin/nagiosstats -c /usr/local/nagios/etc/nagios.cfg
--help
```

This should give you a complete list of the names and meaning of all the fields returned by the output of `nagiosstats`.

3. If you would like to include `mrtg` graphs of this data, a good starting point is using the sample configuration included in the Nagios Core source distribution in `sample-config/mrtg.cfg`.
4. Copy `sample-config/mrtg.cfg` into `/usr/local/nagios/etc`:

```
# cp nagios-4.1.1/sample-config/mrtg.cfg /usr/local/nagios/etc
```

5. Create a directory to store the `mrtg` pages and graphs so that they can be viewed in the Nagios Core web interface:

```
# mkdir /usr/local/nagios/share/stats
```

6. Edit `/usr/local/nagios/etc/mrtg.cfg` to include a `workDir` declaration at the top of the file:

```
WorkDir: /usr/local/nagios/share/stats
```

7. Run `mrtg` to create the graphs:

```
# mrtg /usr/local/nagios/etc/mrtg.cfg
```

For this first run, we can safely ignore any errors about missing prior data or backup log files:

```
2015-11-02 22:03:52, Rateup WARNING: /usr/bin/rateup could not read the
primary log file for nagios-a
2015-11-02 22:03:52, Rateup WARNING: /usr/bin/rateup The backup log
file for nagios-a was invalid as well
2015-11-02 22:03:52, Rateup WARNING: /usr/bin/rateup Can't rename
nagios-a.log to nagios-a.old updating log file
2015-11-02 22:03:52, Rateup WARNING: /usr/bin/rateup could not read the
primary log file for nagios-b
2015-11-02 22:03:52, Rateup WARNING: /usr/bin/rateup The backup log
file for nagios-b was invalid as well
2015-11-02 22:03:52, Rateup WARNING: /usr/bin/rateup Can't rename
nagios-b.log to nagios-b.old updating log file
```

If you are using a UTF-8 locale in your shell, mrtg may fail to run. You can run it in a standard C locale with an env prefix:

```
# env LANG=C mrtg /usr/local/nagios/etc/mrtg.cfg
```

8. Run the indexmaker helper installed with mrtg to create an index on the graphs:

```
# indexmaker /usr/local/nagios/etc/mrtg.cfg --
output=/usr/local/nagios/share/stats/index.html
```

You only need to do this once, unless you add or remove graph definitions from mrtg.cfg later on.

9. Visit <http://olympus.example.net/nagios/stats> and substitute your own Nagios Core server's hostname for olympus.example.net. After authenticating (if necessary), we should be able to see some empty mrtg graphs.



Don't worry that they're empty; we expect that, as we only have one data point for each graph at the moment.

10. If everything has worked up to this point, we can probably add a cron task to run every 5 minutes to add new data points to the graph. Here, we are assuming that the mrtg program is saved in /usr/bin:

```
*/5 * * * * root /usr/bin/mrtg /usr/local/nagios/etc/mrtg.cfg
```

The best way to do this will vary between systems. You could put this in /etc/crontab or in its own file at /etc/cron.d/nagiosstats if you want to be a little tidier.

It is probably safe to leave it running as root, but, if you are concerned about this, you should also be able to run it as the nagios user by including a --lock-file option:

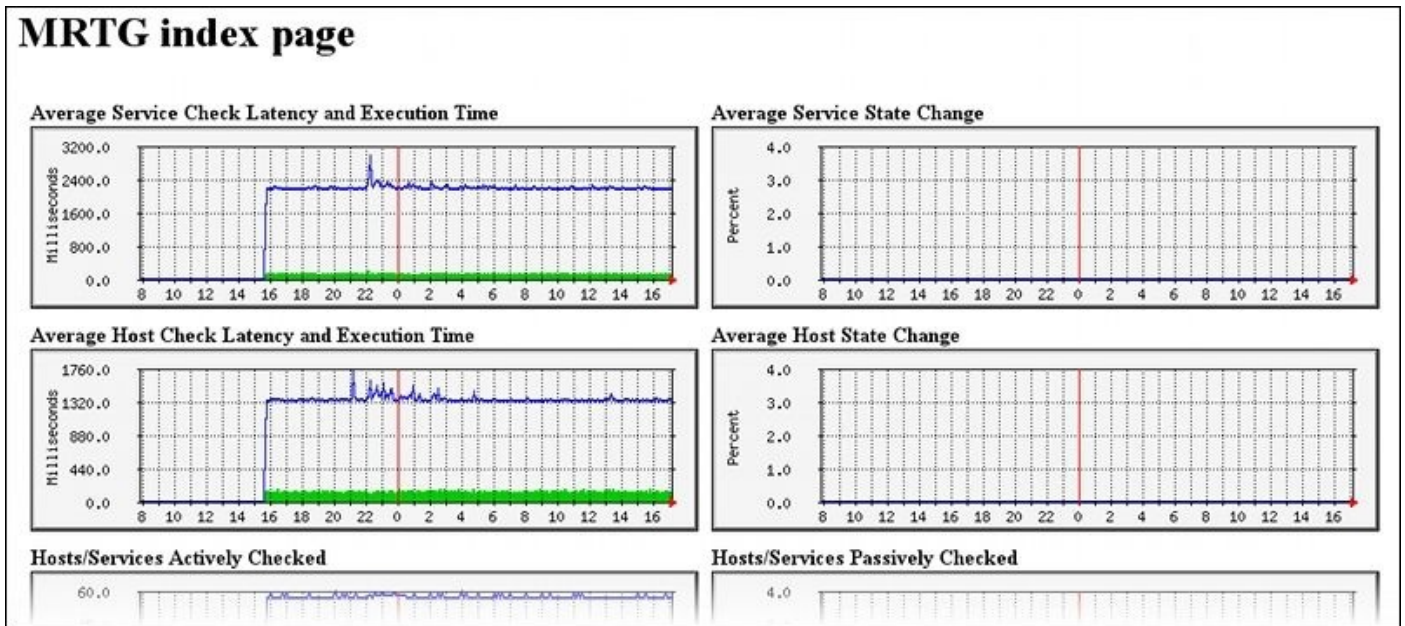
```
*/5 * * * * nagios /usr/bin/mrtg --lock-file=/usr/local/nagios/var/mrtg.cfg.lock /usr/local/nagios/etc/mrtg.cfg
```

This might require us to correct permissions on the graphs that are already generated:

```
# chown -R nagios.nagios /usr/local/nagios/share/stats
```

With this done, if the cron task is installed correctly, we should start seeing data being plotted over the next few hours.

## MRTG index page



## How it works...

The statistics provided by `nagiosstats` provides both performance data about Nagios Core itself, such as how long it takes to complete its round of checks of all objects and the average time it takes per check, as well as data such as the number of hosts in various states. By default, running it will return the data in a terse but human-readable format; you can get a good idea of the meaning of each of the fields by running it with `--help`, as suggested in the recipe.

The `mrtg.cfg` file included in the Nagios source distribution, which is tailored to your particular system at the `./configure` time, contains example definitions of `mrtg` graphs that parse the data retrieved from `nagiosstats`. These are not the only possible graphs using the information provided by `nagiosstats`, but they are useful examples.

The data used is the same as the data you read if you've invoked `nagiosstats` from the shell, but the format is slightly different. If you want to see the data being passed to `mrtg` by `nagiosstats`, you can run it with the `--mrtg` option and nominate fields to be included in the output with `--data`, for example:

```
# /usr/local/nagios/bin/nagiosstats -c /usr/local/nagios/etc/nagios.cfg --  
mrtg --data=AVGACTSVCPSC,AVGPSVSVCPSC,PROGRUNTIME,NAGIOSVERPID  
4  
0  
0d 0h 5m 29s  
Nagios 4.1.1 (pid=22548)
```

The recipe's call to `indexmaker` is a separate program that builds an `index.html` file with links to all the graphs, just for convenience. Like the `mrtg` call, it refers to the `/usr/local/nagios/etc/mrtg.cfg` configuration file to figure out what it needs to do.

## There's more...

Once you're happy with the way your graph's web pages are being displayed, you might like to consider including them in your Nagios Core sidebar. This can be done by editing `/usr/local/nagios/share/side.php` and adding a new link to the **System** section, perhaps named **Performance Reports**, below the **Performance Info** link. The new line might look something like this:

```
<li><a href="/nagios/stats/" target="<?php echo $link_target;?>">Performance Reports</a></li>
```

This would make a link to the graphs show up in the web interface as follows:



### Tip

Note that customizations to a menu like this will be overwritten if you upgrade or reinstall Nagios Core.

If you like what `mrtg` does with this data, you might like to look at Cacti, which is a very helpful frontend to `rrdtool`, which is similar to `mrtg`. It will allow you a lot of flexibility to define graphs, although it takes a while to learn. You can find more information about it at <http://www.cacti.net/>.

Another idea if you're interested in more graphing for Nagios Core performance and state data is the NagiosGraph extension, which is discussed in the *Tracking host and service states with NagiosGraph* recipe in [Chapter 11](#), *Automating and Extending Nagios Core*.

### Tip

Finally, note that Nagios Core includes some built-in graphing of hosts and states in its reports, so be sure to check those out as well before you try to build a graph for a report that already exists! These are all also discussed in [Chapter 7](#), *Using the Web Interface*; check out the references in the *See also* section for this recipe.

## See also

- *Using the tactical overview, [Chapter 7](#), Using the Web Interface*
- *Viewing and interpreting availability reports, [Chapter 7](#), Using the Web Interface*
- *Viewing and interpreting trends, [Chapter 7](#), Using the Web Interface*
- *Viewing and interpreting notification history, [Chapter 7](#), Using the Web Interface*
- *Tracking host and service states with Nagiosgraph, [Chapter 11](#), Automating and Extending Nagios Core*





# Setting up a redundant monitoring host

In this recipe, you'll learn how to implement a simple kind of redundancy for Nagios Core by running a second Nagios Core instance with a near-identical configuration on another machine.

This may seem like it would not need a recipe to implement. It should be reasonably straightforward to simply copy over the configuration for a Nagios Core system and run it concurrently. There are two main problems with this:

- Every problem detected on the network will fire notification events twice. The administrator charged with looking after the pager might well find this unbearable!
- Everything will be checked twice. On smaller networks with simple checks, this may not be too much of a concern, but it could be an issue on larger, busier networks.

This recipe will solve the first problem by configuring the slave monitoring server to suppress notifications until it detects an issue with the master server. In the *There's More* section, we'll discuss extending this solution to solve the second problem as well by preventing the slave server from making checks or sending notifications while the master server is active.

# Getting ready

This is the most complex recipe in this book and one of the longest recipes as it uses a lot of concepts from many other recipes and chapters. To follow it, you will likely need to have a good working knowledge of the following:

- The building blocks of Nagios Core—hosts, services, contacts, commands, plugins, and notifications—are explained in all the recipes from chapters 1 through 4.
- The remote execution via `check_nrpe` is explained in all the recipes in [Chapter 6](#), *Enabling Remote Execution*. The recipe at one point will tell you to install NRPE on the master server to run a specific plugin, so you should learn how to do this first.
- Event handlers and writing to the command file with them is explained in the *Setting up an event handler script* recipe in [Chapter 11](#), *Automating and Extending Nagios Core*.

The event handler scripts, the most complex part of this setup, are fortunately already written for us; we'll show you how to implement them by copying them out of the Nagios Core source package. You'll, therefore, need to have the sources available for your particular version of Nagios Core. If you need to retrieve the sources again, you can download them again from Nagios Core's website, <http://www.nagios.org/>.

The recipe will start by assuming that we have two monitoring servers:

`olympus.example.net` (192.0.2.11), which will be the master monitoring server, and `everest.example.net` (192.0.2.12), which will be the slave. The two servers are configured to monitor the same three hosts, with PING service checks:

- `sparta.example.net`
- `athens.example.net`
- `ithaca.example.net`

The Nagios Core configuration of the two servers is completely identical to start with and both send notifications to an appropriate contact group.

## Tip

However, note that the servers are not yet monitoring one another; this will be an important part of the recipe.

# How to do it...

We can arrange a simple redundancy setup for our two Nagios Core servers as follows:

1. Confirm that the `check_nagios` plugin is available on the master server and try running it:

```
# cd /usr/local/nagios/libexec
# ./check_nagios -e 5 -F /usr/local/nagios/var/status.dat -C
/usr/local/nagios/bin/nagios
NAGIOS OK: 6 processes, status log updated 3 seconds ago
```

2. Install the NRPE daemon on the master server and define the `check_nagios` command in `nrpe.cfg` (refer to [Chapter 6, Enabling Remote Execution](#)):

```
command[check_nagios]=/usr/local/nagios/libexec/check_nagios -e 5 -F
/usr/local/nagios/var/status.dat -C /usr/local/nagios/bin/nagios
```

3. Include the slave server's address in the `allowed_hosts` directive for `nrpe` in the `/usr/local/nagios/etc/nrpe.cfg` file:

```
allowed_hosts=127.0.0.1,192.0.2.12
```

Don't forget to restart NRPE to include this change to the configuration.

4. On the slave server, verify that a call to `check_nrpe` can retrieve the results of `check_nagios` on the master server:

```
# cd /usr/local/nagios/libexec
# ./check_nrpe -H olympus.example.net
NRPE v2.15
# ./check_nrpe -H olympus.example.net -c check_nagios
NAGIOS OK: 1 process, status log updated 2 seconds ago
```

You will have to install the `check_nrpe` plugin on the slave server to do this. This is explained in the *Monitoring local services on a remote machine with NRPE* recipe in [Chapter 6, Enabling Remote Execution](#).

5. On the slave server, copy 4 files (2 event handlers and 2 helper scripts) from the source distribution into the `/usr/local/nagios/libexec/eventhandlers` directory (which you may need to create first):

```
# ehd=/usr/local/nagios/libexec/eventhandlers
# mkdir -p "$ehd"
# cd /usr/local/src/nagios
# cp contrib/eventhandlers/enable_notifications "$ehd"
# cp contrib/eventhandlers/disable_notifications "$ehd"
# cp contrib/eventhandlers/redundancy-scenario1/handle-master-host-
event "$ehd"
# cp contrib/eventhandlers/redundancy-scenario1/handle-master-proc-
event "$ehd"
```

The preceding step assumes that you are keeping the sources for your Nagios Core distribution in `/usr/local/src`. We define and use the shell variable `ehd` to refer to the event handlers directory for convenience.

6. In the installed `handle-master-proc-event` script, find and replace `active_service_checks` with `notifications`. The command-line tool `ed(1)` works well for this:

```
# printf '%s\n' ,s/active_service_checks/notifications/g w | ed -s
-"$ehd"/handle-master-proc-event
```

This is because the script, as provided, issues a command to toggle active checks, rather than notifications. At the time of writing, in Nagios 4.1.1, there is also a bug in `handle-master-proc-event`, which will need to be corrected, on line 49:

```
`eventhandlerdir/disable_active_service_checks`
```

It should have a dollar sign added after the first backtick:

```
`$eventhandlerdir/disable_active_service_checks`
```

7. Ensure that the event handlers are owned and executable by the `nagios` user:

```
# chown nagios.nagios "$ehd"/*
# chmod 0755 "$ehd"/*
```

8. In `/usr/local/nagios/etc/objects/commands.cfg`, define two new event handler commands:

```
define command {
    command_name    handle-master-host-event
    command_line    $USER1$/eventhandlers/handle-master-host-event
$HOSTSTATE$ $HOSTSTATETYPE$ $HOSTATTEMPT$
}
define command {
    command_name    handle-master-proc-event
    command_line    $USER1$/eventhandlers/handle-master-proc-event
$SERVICESTATE$ $SERVICESTATETYPE$ $SERVICEATTEMPT$
}
```

9. Make a host and service definition on the slave server to monitor the master server. It might look something like this; change your `host_name`, `alias`, and `address` values as appropriate. The templates used are only examples; you will probably want to choose templates that are defined to run checks often and during a 24x7 interval:

```
define host {
    use                critical-host-template
    host_name          olympus.example.net
    alias              olympus
    address             192.0.2.11
    event_handler      handle-master-host-event
}
define service {
    use                critical-service-template
    host_name          olympus.example.net
    service_description NAGIOS
    check_command       check_nrpe!check_nagios
    event_handler      handle-master-proc-event
}
```

You can make the master server monitor the slave server as well if you wish, but it is not necessary for this redundancy setup.

- Note that you will need to have the `check_nrpe` command defined, which is explained in the *Monitoring local services on a remote machine with NRPE* recipe in [Chapter 6, Enabling Remote Execution](#). If you have followed that recipe, then you have probably already done this. If not, the following definition works:

```
define command {  
    command_name    check_nrpe  
    command_line    $USER1$/check_nrpe -H $HOSTADDRESS$ -c $ARG1$  
}
```

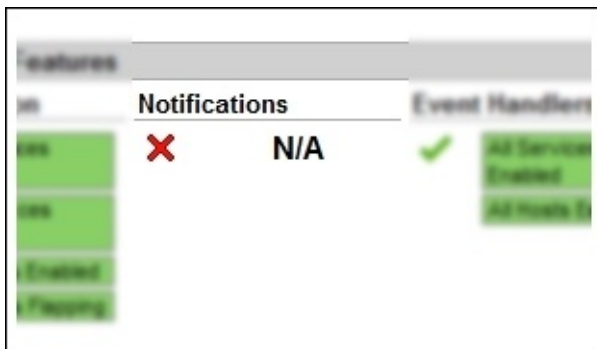
- Finally, in `nagios.cfg`, on the slave server, change `enable_notifications` to 0:

```
enable_notifications=0
```

- Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg  
# /etc/init.d/nagios restart
```

With this done, the two Nagios Core servers should both be running, but, importantly, notifications on the slave server start out as disabled, as visible in the Tactical Overview.



However, all the systems are still being monitored, as visible in the following **Services** screen, including the NAGIOS service on the host machine:

| Host ↕              | Service ↕ | Status ↕ |
|---------------------|-----------|----------|
| athens.example.net  | PING      | OK       |
| ithaca.example.net  | PING      | OK       |
| olympus.example.net | NAGIOS    | OK       |
| sparta.example.net  | PING      | OK       |

This means that notifications will only be sent by the master server, since it still has its notifications enabled. However, if the master server goes down or its Nagios process stops working, the event handlers should be called and notifications on the slave server will be automatically enabled. When the master server or its NAGIOS service comes back up, the

notifications will be disabled again, with checks and state changes having continued uninterrupted throughout. We have therefore established a simple kind of redundancy. If you use this setup, you should test it thoroughly to make sure that the slave Nagios Core server will enable and disable its notifications for each contingency (host goes down, service goes down, service comes back, and so on.)

# How it works...

The event handlers included in the Nagios Core distribution, which we copied into the `eventhandlers` directory, are designed to handle toggling notifications and active checks based on the status of a given service or host. They are included for the purposes of demonstrating event handlers and redundancy situations like this one.

We start by setting up the slave server to monitor not just the host on which the master Nagios Core server is running, but also the Nagios Core service itself, using the `check_nagios` plugin. This plugin checks the age of the log file and the system's process tables to ensure that a Nagios Core service is actually running on the system. Because it's a local plugin that doesn't work for remote checks, we check it from the slave server via NRPE.

The slave server checks the status of the master server and its NAGIOS service as part of its normal routine of active checks. When the master server's host or its NAGIOS service change state, both call their respective event handlers, the two shell scripts `handle-master-host-event` and `handle-master-proc-event` that are defined in the commands of the same name.

Each time the event handlers are called, they are passed three arguments in a macro form. For `handle-master-host-event`, these are:

- `$HOSTSTATE$`: This argument is the new state of the master server
- `$HOSTSTATETYPE$`: This argument shows whether the state is `SOFT` or `HARD`
- `$HOSTATTEMPT$`: This argument shows the number of host checks attempted up to the value of `max_check_attempts` for the host

The `handle-master-proc-event` is passed three analogous arguments, the only difference being they refer to service states rather than host states:

- `$SERVICESTATE$`: This argument is the new state of the NAGIOS service on the master server
- `$SERVICESTATETYPE$`: This argument shows whether the service state is `SOFT` or `HARD`
- `$SERVICEATTEMPT$`: This argument shows the number of host checks attempted, up to the value of `max_check_attempts` for the host

The event handlers are written in such a way that they only do anything if the new state is `HARD`, that is, if the number of `max_check_attempts` has been reached. It ignores `SOFT` state changes until enough consecutive checks have failed that it can be reasonably confident in concluding that the monitored host or service is suffering a problem.

If the host or service enters a `HARD CRITICAL` state, the event handlers call the helper script `enable-notifications` to write a command to the commands file at `/usr/local/nagios/var/rw/nagios.cmd` for the server to process. This command takes the following form, including the Unix timestamp for when the command was written:

```
[1446457199] ENABLE_NOTIFICATIONS;1446457199
```

When Nagios Core processes this command, the effect is that the previously disabled



notifications are enabled and all subsequent notifications generated as a result of checks will be sent.

Similarly, when the host or service recovers from the `HARD CRITICAL` state, by entering a `HARD UP` or `HARD OK` state, the `disable-notifications` helper script is called, writing a command in the same manner:

**[1446457218] DISABLE\_NOTIFICATIONS;1446457218**

The effect is that when the master server is noted to be down, the slave server notices and assumes its notification behavior and, when it recovers, it stops its own notifications again, allowing the master server to resume its role.

## There's more...

If network bandwidth or other resources are a concern, we can arrange to leave the slave server more or less idle when not in use. We can do this by keeping not only notifications, but also service checks, off by default. Helper scripts for this are also included in the Nagios Core distribution in the `disable_active_service_checks` and `enable_active_service_checks` scripts.

The primary issue with this change is the loss of state information, as the slave server makes its initial round of checks; this can also be worked around, as explained in the Nagios Core documentation on redundancy:

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/redundancy.html>

Once these steps are implemented, the main issue is to keep the two configuration directories in sync. It's undesirable and error prone to have to make changes on two servers each time the configuration needs to change, so you may like to consider using a snapshot tool such as `rsync` to keep the two directories the same:

<http://en.wikipedia.org/wiki/Rsync>

Even better is a configuration managed with version control, which can also help here, as recommended in the *Keeping configuration under version control* recipe in [Chapter 9, Managing Configuration](#). This way you can use `git clone` or `svn checkout` to quickly update configuration files on multiple machines.

## See also

- *Monitoring local services on a remote machine with NRPE, [Chapter 6](#), Enabling Remote Execution*
- *Keeping configuration under version control, [Chapter 9](#), Managing Configuration*
- *The Writing debugging information to Nagios log file recipe in this chapter*
- *Setting up an event handler script, [Chapter 11](#), Automating and Extending Nagios Core*



# Chapter 11. Automating and Extending Nagios Core

In this chapter, we will cover the following recipes:

- Allowing and submitting passive checks
- Submitting passive checks from a remote host with NSCA
- Submitting passive checks in response to SNMP traps
- Setting up an event handler script
- Tracking host and service states with Nagiosgraph
- Reading status in a MySQL database with NDOUtils
- Reading status from a Unix socket with MK Livestatus
- Writing customized Nagios Core reports
- Getting extra visualizations with NagVis
- Writing custom Nagios Core management scripts

# Introduction

In addition to being useful as a standalone monitoring framework, Nagios Core has a modular design that allows both interaction with and extension to other programs and tools. It does so predominantly using its **external command file** to control the behavior of the server.

One of the most useful ways of interacting with the Nagios Core server in this way is through the use of **passive checks** or by checking results for hosts and services that are submitted to the server directly, rather than as the result of the server's own **active checks**.

The simplest application of the idea of passive checks is to monitor a process that might take an indeterminate amount of time to run and hence resists active checking; instead of the service making active checks of its own, the service accepts a check result submitted by another application, perhaps something like a backup script after it has completed its run. These check results can be sent and accepted via an add-on called **NSCA** (the **Nagios Service Check Acceptor**). Similarly, just as plugins and notifications are actually scripted calls to external commands such as `check_http` and `mail`, **event handlers** can be configured to run a specified command every time a host or service changes state. This can be used for supplementary recording of how hosts and services change state, or automated attempts at actively resolving a problem, such as restarting a remote server. We also saw event handlers used in the redundant monitoring recipe *Setting up a redundant monitoring host* in [Chapter 10, Security and Performance](#).

Finally, this chapter also includes installation procedures and discussions of a few of the more popular extensions of Nagios Core, which are as follows:

- **Nagiosgraph:** This is the advanced web-based graphing solution for Nagios Core that graphs both server performance and the status metrics of a host and service.
- **NagVis:** This refers to advanced web-based visualizations for Nagios Core data. This is especially well suited for administrators who need something more comprehensive than Nagios Core's built-in network mapping.
- **NDOUtils:** This is used to apply translations of Nagios Core data into a standard database system such as MySQL; this is very useful for performing advanced queries of Nagios Core data for custom systems such as to monitor displays or to log in with change control systems.
- **MK Livestatus:** Part of Mathias Ketner's Check\_MK suite, this allows direct access to Nagios Core data via a query language written to a Unix socket; this tends to be faster than NDOUtils and doesn't require a database server.

We'll discuss NDOUtils and MK Livestatus, perhaps the most versatile Nagios Core extensions of all, in three separate recipes; first, we'll show you how to install and use each extension and then we'll share some ideas on how to apply NDOUtils to make custom Nagios Core reporting applications of our own in the form of a CLI report written in Perl and an RSS feed written in PHP.



# Allowing and submitting passive checks

In this recipe, you'll learn how to configure Nagios Core to accept passive checks for a service. This allows both users and external applications to directly submit the results of checks to Nagios Core, rather than have the application seek them out itself through polling with active checks, which are performed via plugins such as `check_http` or `check_ping`.

We'll show you one simple example of a passive check: flagging a service called `BACKUP` for an existing host. We'll show you how to do this via a web interface, which is very easy, and via the external commands file, which is slightly more complex but much more flexible and open to automation.

The idea is that when a user or process receives confirmation that the backup process on a host has completed correctly, the user or process is able to supply a check result of `OK` directly to the service without Nagios Core needing to poll anything by itself.



# Getting ready

You should be running a Nagios Core 4.0 or newer server. You should also already have a host configured for which you want to define a service that will accept passive checks. In this example, we'll use the `ithaca.example.net` host, which can be defined as follows:

```
define host {  
    use          linux-server  
    host_name    ithaca.example.net  
    alias        ithaca  
    address      192.0.2.21  
}
```

You will also need a working Nagios Core web interface to check whether passive checks are enabled and to try out the recipe's method of submitting passive checks.

This recipe will be presented in two parts: in the first part we will enable and configure the service for passive checks only and then we'll actually submit a passive check via the web interface. In the *There's more* section, we'll show you how to submit a check result via the external commands file, which is a little more complicated but allows advanced automation behavior.

# How to do it...

We can define a new BACKUP service that accepts only passive checks as follows:

1. Log in to the web interface and ensure that the passive checks are enabled. The **Tactical Overview** section shows a panel for it near the bottom of the screen. Verify that it's green.



If it's not green, you should be able to enable the checks again by clicking on the cross mark, where the tick should be. In this case, you should also check the `/usr/local/nagios/etc/nagios.cfg` file to make sure that the `accept_passive_service_checks` option is set to 1 so that Nagios Core allows passive checks on startup.

2. Change to the Nagios Core object's configuration directory. If you're using a sample configuration, this directory will likely be `/usr/local/nagios/etc/objects`:

```
# cd /usr/local/nagios/etc/objects
```

3. Edit the `commands.cfg` file and add a definition for the `check_dummy` command:

```
define command {
    command_name    check_dummy
    command_line    $USER1$/check_dummy $ARG1$ $ARG2$
}
```

If you already followed the *Monitoring individual nodes in a cluster* recipe in [Chapter 8, Managing Network Layout](#), you may already have defined this command, in which case, you can skip this step, as the definition is the same.

4. Edit the file containing the definition for the existing host. In this example, the host is defined in a file called `ithaca.example.net.cfg`:

```
# vi ithaca.example.net.cfg
```

5. Add the following service definition to the end of the file, substituting the appropriate value for `host_name`:

```
define service {
    use                generic-service
    host_name          ithaca.example.net
    service_description BACKUP
    active_checks_enabled 0
    passive_checks_enabled 1
    check_command      check_dummy!1!"Unwanted active check!"
}
```

This example uses the generic-service template. You can use any service template you like; the important directives are `active_checks_enabled`, `passive_checks_enabled`, and `check_command`.

6. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, the Nagios Core web interface should show that the service is accepting passive checks only in the **Services** listing, as shown in the following screenshot:

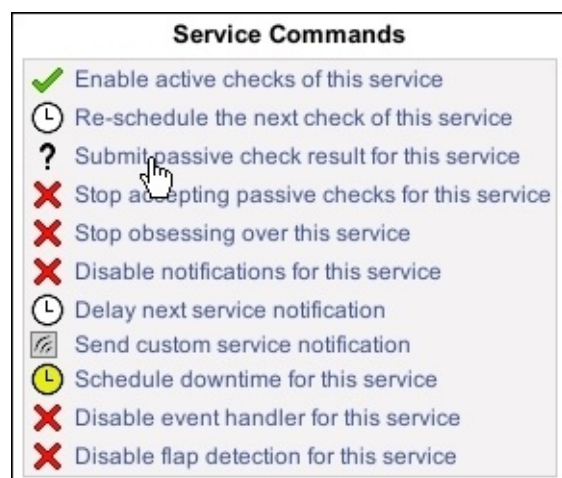
| Host ↕             | Service ↕ | Status ↕ | Last Check ↕ | Duration ↕   | Attempt ↕ | Status Information                     |
|--------------------|-----------|----------|--------------|--------------|-----------|----------------------------------------|
| ithaca.example.net | BACKUP    | PENDING  | N/A          | 0d 0h 0m 4s+ | 1/3       | Service is not scheduled to be checked |
|                    | HTTP      |          |              |              |           | HTTP OK: HTTP/1.1 200 OK - 1           |
|                    | PING      |          |              |              |           |                                        |

Active checks of the service have been disabled - only passive checks are being accepted

The service will remain in the PENDING state until a passive check result is submitted for it.

We can submit a passive check result via the web interface as follows:

1. Click on the service's name in the **Services** listing and click on **Submit passive check result for this service** menu item:



2. Complete the resulting form with the following values:

- **Host Name:** This is the hostname for which the passive check result should be submitted. This should already be filled out for us.
- **Service:** This refers to the service description for which the passive check result should be submitted. This should also already be filled out with BACKUP.
- **Check Result:** This refers to the particular result you would like to submit for the check. In this case, we've chosen OK to signal that the backup has completed successfully. We could just as easily submit a CRITICAL result if we wished.
- **Check Output:** This is a message that is to be attached to the status. In this

case, we choose a simple message: **Nightly backups were successful.**

- **Performance Data:** This refers to the optional extra details about how the service being checked performs. We can leave this blank.

3. Click on **Commit** to submit the passive check result.

| Command Options                                                            |                                                         |
|----------------------------------------------------------------------------|---------------------------------------------------------|
| Host Name:                                                                 | <input type="text" value="ithaca.example.net"/>         |
| Service:                                                                   | <input type="text" value="BACKUP"/>                     |
| Check Result:                                                              | <input type="button" value="OK"/>                       |
| Check Output:                                                              | <input type="text" value="Nightly backups were suces"/> |
| Performance Data:                                                          | <input type="text"/>                                    |
| <input type="button" value="Commit"/> <input type="button" value="Reset"/> |                                                         |

After a short delay, the details for the service should show the result of the passive check as reflecting the result of the passive check. Along with this, it will explicitly show that active checks are disabled.

| Service State Information |                                        |
|---------------------------|----------------------------------------|
| Current Status:           | <b>OK</b> (for 0d 0h 0m 4s)            |
| Status Information:       | Nightly backups were sucessful         |
| Performance Data:         |                                        |
| Current Attempt:          | 1/3 (HARD state)                       |
| Last Check Time:          | 11-25-2015 15:15:34                    |
| Check Type:               | PASSIVE                                |
| Check Latency / Duration: | N/A / 0.000 seconds                    |
| Next Scheduled Check:     | N/A                                    |
| Last State Change:        | 11-25-2015 15:15:34                    |
| Last Notification:        | N/A (notification 0)                   |
| Is This Service Flapping? | <b>NO</b> (0.00% state change)         |
| In Scheduled Downtime?    | <b>NO</b>                              |
| Last Update:              | 11-25-2015 15:15:36 ( 0d 0h 0m 2s ago) |
| Active Checks:            | <b>DISABLED</b>                        |
| Passive Checks:           | <b>ENABLED</b>                         |
| Obsessing:                | <b>ENABLED</b>                         |
| Notifications:            | <b>ENABLED</b>                         |
| Event Handler:            | <b>ENABLED</b>                         |
| Flap Detection:           | <b>ENABLED</b>                         |

## How it works...

The preceding configuration adds a new service called `BACKUP` to the existing `ithaca.example.net` host and is designed to manage and report the status of backups for this host. In our example, this isn't something Nagios Core can check manually; no network service on `ithaca.example.net` can be checked to verify that the backups have succeeded.

However, suppose as administrators we do receive a backup report in our inbox every morning. We come to know whether the backups have succeeded or failed and would like to register that status in Nagios Core, perhaps for record-keeping purposes or to alert other administrators to problems.

To that end, we disable active checks for the service and put in place a dummy check command, `check_dummy` that we never expect to run. If, for whatever reason, an active check is run, it will always flag a `WARNING` state with the “Unwanted active check!” message. The `check_dummy` command never actually checks anything; it is configured always to return the state and output defined in its two arguments.

Instead, we enable passive checks for the service and submit the results manually. If the backups fail, we could just as easily record the same with a passive check result of `WARNING` or `CRITICAL`.

## There's more...

It's also possible (and often desirable) to submit active checks via the external commands file, which is useful for automation purposes. We write details for the check in a single line into the commands file in the following format:

```
[<timestamp>] PROCESS_SERVICE_CHECK_RESULT;<host_name>;  
<service_description>;<service_status>;<plugin_output>
```

For our example, the line would be something like this:

```
[1348806599]  
PROCESS_SERVICE_CHECK_RESULT;ithaca.example.net;BACKUP;0;Nightly backups  
were successful
```

We can write this directly to the external commands file as follows:

```
# check="[$(date +%s)]  
PROCESS_SERVICE_CHECK_RESULT;ithaca.example.net;BACKUP;0;Nightly backups  
were successful"  
# printf '%s\n' "$check" >>/usr/local/nagios/var/rw/nagios.cmd
```

In this context, the `<service_status>` field needs to be an integer corresponding to the appropriate state. If you use the text value OK or WARNING, the command will not work as expected. The values to use for this field are:

- 0 for OK
- 1 for WARNING
- 2 for CRITICAL
- 3 for UNKNOWN

If the syntax is correct, then the passive check will be registered in just the same way as if it were submitted via a web interface. Writing to the command file thus allows us to submit passive check results with scripts and automated systems, with a little knowledge of an appropriate scripting language such as Bash or Perl.

We will go into a little more detail about using external commands for passive check results, including a common application with the NSCA add-on in the *Submitting passive checks from a remote host with NSCA* recipe in this chapter. If you don't want to submit your passive checks manually, then you will most likely find this recipe interesting along with its accompanying explanation of freshness checks.

## See also

- The *Submitting passive checks from a remote host with NSCA* recipe in this chapter
- The *Submitting passive checks in response to SNMP traps* recipe in this chapter
- The *Setting up an event handler script* recipe in this chapter





# Submitting passive checks from a remote host with NSCA

In this recipe, we'll show you how to automate the submission of passive checks by a remote host using the example of a monitored host, `ithaca.example.net`, to submit a passive check to a Nagios Core server with information about how its `BACKUP` service is performing.

For example, if the backup process gets completed successfully, we configure the monitored host to submit a passive check result, specifying that the `BACKUP` service should have the `OK` status. However, if there were a problem with the backup, the monitored host could send a passive check result with a `WARNING` or `CRITICAL` status.

In both cases, Nagios Core does no checking of its own; it trusts the results submitted by its target host.

To accomplish this setup, we'll use the NSCA add-on. We'll install the NSCA server on the Nagios Core server and the NSCA client program `send_nsca` on the monitored host.

# Getting ready

Here, we assume that you have already followed the *Allowing and submitting passive checks* recipe in this chapter. In this recipe, we will build on the configuration established in that recipe; specifically, we will assume that you already have a host with a service configured only to accept passive checks.

You will need to be able to install software on both the monitoring server (the NSCA server) and on the server that will submit passive checks (the NSCA client) and ideally be generally familiar with the `./configure`, `make`, and `make install` process to install software from source on Unix-like systems.

You should also be able to define any necessary firewall configuration to allow the NSCA client to send information to TCP port 5667 on the NSCA server. A firewall is absolutely necessary to protect the `nsca` daemon from abuse.

# How to do it...

We can set up the NSCA server on the monitoring server (in this example, `olympus.example.net`) as follows:

1. Download the latest version of NSCA using `wget` (1) or a similar tool. You can find download links on the Nagios Exchange page for NSCA at <https://exchange.nagios.org/directory/Addons/Passive-Checks/NSCA—2D-Nagios-Service-Check-Acceptor/details>.

In this example, we're downloading and compiling the NSCA sources in our home directory on the monitoring server:

```
$ cd
$ wget http://downloads.sourceforge.net/project/nagios/nsca-2.x/nsca-2.9.1/nsca-2.9.1.tar.gz
```

2. Inflate the `.tar.gz` file:

```
$ tar xzf nsca-2.9.1.tar.gz
```

3. Move into the new `nsca-2.9.1` directory and configure and compile the `nsca` daemon. Note that this process may prompt you to install the `libmcrypto` library and its headers, perhaps in `libmcrypto` and `libmcrypto-dev` packages in your system's package manager:

```
$ cd nsca-2.9.1
$ ./configure
$ make
```

4. Install the NSCA server files manually; you will likely need root privileges for this:

```
# cp src/nsca /usr/local/nagios/bin
# cp sample-config/nsca.cfg /usr/local/nagios/etc
```

5. Edit the new file, `/usr/local/nagios/etc/nsca.cfg`:

```
# vi /usr/local/nagios/etc/nsca.cfg
```

6. Uncomment the password directive and define it. A random password generated by a tool such as `pwgen` or `makepasswd` will work fine. Don't use the following one—it's just an example!

```
password=yV3aa6S2o
```

7. Verify that the NSCA daemon runs with no errors:

```
# /usr/local/nagios/bin/nsca -c /usr/local/nagios/etc/nsca.cfg --single
```

If it does, you should add this command to an appropriate startup script, perhaps in `/etc/rc.local` so that the daemon starts when the monitoring server boots. You should consult your system's documentation to find out the best place to add this. A templated `init(8)` script for `/etc/init.d` should be built with the distribution in the `init-script` file.

We can set up the NSCA client on the monitored server (in this example, `ithaca.example.net`) as follows:

1. Again, download and expand the latest version of NSCA and configure and compile it:

```
$ cd
$ wget http://downloads.sourceforge.net/project/nagios/nsca-2.x/nsca-2.9.1/nsca-2.9.1.tar.gz
$ tar xzf nsca-2.9.1.tar.gz
$ cd nsca-2.9.1
$ ./configure
$ make
Install the NSCA client files manually; you will likely need root
privileges for this:
# cp src/send_nsca /usr/local/bin
# cp sample-config/send_nsca.cfg /etc
```

2. Edit the `/etc/send_nsca.cfg` new file:

```
# vi /etc/send_nsca.cfg
```

3. Uncomment the password directive and define it such that it is the same as the password given in `nsca.cfg` on the monitoring server:

```
password=yV3aa6S2o
```

4. Run the `send_nsca` program to try and submit a passive check result:

```
# check="ithaca.example.net\tBACKUP\t0\tBackup was successful, this
check submitted by NSCA\n"
# printf "$check" | send_nsca -c / etc/send_nsca.cfg -H
olympus.example.net
1 data packet(s) sent to host successfully.
```

Substitute the appropriate hostnames for the monitoring server (`olympus.example.net`), the monitored server (`ithaca.example.net`), and the service description `BACKUP`.

Note that the fields are separated by `\t` characters, which expand to literal tab characters with `printf`.

If this worked correctly, you should see in the web interface that the passive check result was successfully read by the monitoring server and was applied appropriately.

### Service State Information

**Current Status:** **OK** (for 0d 0h 16m 16s)  
**Status Information:** Backup was successful, this check submitted by NSCA  
**Performance Data:**  
**Current Attempt:** 1/3 (HARD state)  
**Last Check Time:** 11-25-2015 15:31:14  
**Check Type:** PASSIVE  
**Check Latency / Duration:** N/A / 0.000 seconds  
**Next Scheduled Check:** N/A  
**Last State Change:** 11-25-2015 15:15:34  
**Last Notification:** N/A (notification 0)  
**Is This Service Flapping?** **NO** (0.00% state change)  
**In Scheduled Downtime?** **NO**  
**Last Update:** 11-25-2015 15:31:46 ( 0d 0h 0m 4s ago)

**Active Checks:** **DISABLED**  
**Passive Checks:** **ENABLED**  
**Obsessing:** **ENABLED**  
**Notifications:** **ENABLED**  
**Event Handler:** **ENABLED**  
**Flap Detection:** **ENABLED**

# How it works...

The nsca daemon installed on the monitoring server is designed to listen for submitted service checks from the send\_nsca client, provided the password is correct and the data is in the appropriate format:

```
<host_name>\t<service_description>\t<check_result>\t<check_output>\n
```

Our example passive check took this form:

```
ithaca.example.net\tBACKUP\t0\tBackup was successful, this check submitted  
by NSCA\n
```

Here, as with locally submitted passive checks, the check\_result command corresponds to a numeric value, which is one of the following values:

- 0 for OK
- 1 for WARNING
- 2 for CRITICAL
- 3 for UNKNOWN

Once received by the nsca daemon on the monitoring server, this is translated into a passive check result command that is written to the Nagios Core external commands file at /usr/local/nagios/var/rw/nagios.cmd and processed in the same way as a locally submitted passive check would be.

This allows us to include calls to send\_nsca at the end of scripts, such as those managing backups, to immediately and automatically send a passive check result corresponding to whether the backup script succeeded.

Because of the NSCA daemon's very simple design and very basic security checks, it's important to apply a firewall policy to ensure that only the appropriate hosts can write to the NSCA port on the host monitoring system. A password, as implemented here, is a good first step, but it is not sufficient to keep things secure. Make sure you read the SECURITY file included in the NSCA sources to ensure that your configuration for the daemon is secure. Similar security guidelines apply to the installation of NRPE, as discussed in [Chapter 6](#), *Enabling Remote Execution*.

## There's more...

To supplement this setup, it's often a good idea to also have Nagios Core check the freshness of its services. If we have a process that needs to run regularly, such as backups, we will likely want to be notified if we haven't received any passive checks from the host in a given period of time.

This can be managed by configuring the service to run an active check after a certain period of time has elapsed with no passive checks. The configuration might look like this, adding values for `check_freshness` and `freshness_threshold`:

```
define service {
    use                generic-service
    host_name          ithaca.example.net
    service_description BACKUP
    active_checks_enabled 0
    passive_checks_enabled 1
    check_freshness      1
    freshness_threshold 86400
    check_command       check_dummy!1!"No backups have run for 24
hours"
}
```

In this case, `freshness_threshold` is 86400 seconds, or 24 hours; if there have been no passive checks submitted for 24 hours, `check_command` will be run even though active checks are disabled. The `check_command` is defined to flag a `WARNING` state for the service with an appropriate explanatory message using the `check_dummy` command and plugin whenever it is actually run.

Check freshness is discussed in more detail in the Nagios Core documentation in the section entitled **Service and Host Freshness Checks**, which you can find at <https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/freshness.html>.

Note that there's no reason that the status of a service has to come from the same host. You can send a passive check from one host to submit information about another host. In fact, this is the basis of a **distributed monitoring setup**; one host can submit check results for any number of other hosts.

### Tip

This can be particularly useful for working around network connectivity or routing problems; if Nagios Core has no connectivity at all to a host it needs to monitor, but does have connectivity to an intermediate host, that host can be configured to submit checks on behalf of the unreachable host, which is a slightly complex but often necessary setup.

## See also

- *Using an alternative check command*, [Chapter 2](#), *Working with Commands and Plugins*
- The *Allowing and submitting passive checks* recipe in this chapter
- The *Submitting passive checks in response to SNMP traps* recipe in this chapter
- The *Setting up an event handler script* recipe in this chapter





# Submitting passive checks in response to SNMP traps

In this recipe, you'll learn how to configure Nagios Core to process SNMP traps, which is information sent by monitored network devices to a central monitoring server.

Because SNMP traps often contain useful or urgent information about how a host is working, processing them in at least some way can be very helpful. It can be very helpful particularly for firmware network devices that can't use `send_nsca` to submit a passive check result in a standard form, as explained in the *Submitting passive checks from a remote host with NSCA* recipe.

As an example, most SNMP-capable hosts can be configured to send SNMP traps when one of their network interfaces changes state, perhaps due to a pulled network cable. These traps are known as `linkUp` and `linkDown` traps. Monitoring this particular kind of trap is especially useful for devices with a large number of interfaces, such as switches or routers.

Keeping track of these events in Nagios Core is valuable in order to keep a unified monitoring interface, as opposed to monitoring SNMP traps with a separate application.

# Getting ready

There are quite a few prerequisites to getting this recipe to work. This method is among the most powerful but also most complex methods of Nagios Core monitoring.

First of all, the recipe assumes some knowledge of the Simple Network Management Protocol (SNMP). Unfortunately, SNMP is not very simple, despite its name! You should be familiar with the concepts of SNMP checks and SNMP traps. The documentation for **Net-SNMP** (the implementation of SNMP used for this example) may help here, which you can find at <http://www.net-snmp.org/docs/readmefiles.html>.

On the same host as your Nagios Core server with version 4.0 or greater, you should have `snmptrapd` installed to collect trap information and `snmptt`, the SNMP Trap Translator, to filter useful information from the traps and submit the information to Nagios Core in a workable format. The documentation for SNMPTT is available at <http://snmptt.sourceforge.net/docs/snmptt.shtml>.

Both systems are free software and relatively popular, so check whether there are packages available for your particular system to save the hassle of compiling them from source. On Debian-derived systems, such as Ubuntu, for example, they are available in the `snmpd` and `snmptt` packages.

We will use an event handler called `submit_check_result`, which is available in the Nagios Core distribution. You will, therefore, need to have access to the original sources handy. If you have misplaced them, you can download them again from Nagios' website (<https://www.nagios.org/downloads/>).

It's also necessary to use values of `host_name` for your hosts that actually correspond to hostnames resolvable by DNS from the monitoring server. This is because when the SNMP trap is received by SNMPTT, the only way it can translate the trap's source IP address to a hostname is with DNS. The `host_name` value for your host might be `crete.example.net`, but the trap will arrive from an IP address like `192.0.2.27`. The system will, therefore, need to be able to resolve this with a reverse DNS lookup. An easy way to test whether this is working is to use `host(1)` or `dig(1)`:

```
$ host 192.0.2.27
27.2.0.192.in-addr.arpa domain name pointer crete.example.net.
$ dig -x 192.0.2.27 +short
crete.example.net.
```

Finally, you should, of course, actually have a device configured to send SNMP traps to your monitoring server, which in turn is configured to listen for SNMP traps with the `snmpd` daemon. I don't really want to encourage you to unplug one of your core switches to test this, so we'll generate a trap manually with `snmptrap(1)` on the monitored server to demonstrate the principle.

# How to do it...

We can configure a new service to receive SNMP traps for an existing host using the following steps:

1. On the monitoring server running Nagios Core, copy the contrib/eventhandlers/submit\_check\_result event handler script from the Nagios Core source files into /usr/local/nagios/libexec/eventhandlers. You may need to create the target directory first. Your source files need not be in /usr/local/src/nagios; this is just an example:

```
# mkdir -p /usr/local/nagios/libexec/eventhandlers
# cp /usr/local/src/nagios/contrib/eventhandlers/submit_check_result
/usr/local/nagios/libexec/eventhandlers
# chmod +x /usr/local/nagios/libexec/eventhandlers/submit_check_result
```

2. Change to the Nagios Core objects configuration directory. For the default configuration, this directory is /usr/local/nagios/etc/objects:

```
# cd /usr/local/nagios/etc/objects
```

3. Edit the file containing the definition for the SNMP-enabled monitored host. In this example, the definition of crete.example.net is in its own crete.example.net.cfg file:

```
# vi crete.example.net.cfg
```

The host definition might look something like this:

```
define host {
    use          linux-server
    host_name    crete.example.net
    alias        crete
    address      192.0.2.27
}
```

4. Add a service definition for your existing host that accepts only passive checks and is flagged as **volatile**. Here, we have used the generic-service template included in the sample configuration. You may prefer to use a different template, but all the values defined here are important:

```
define service {
    use          generic-service
    host_name    crete.example.net
    service_description TRAP
    is_volatile  1
    check_command check-host-alive
    active_checks_enabled 0
    passive_checks_enabled 1
    max_check_attempts 1
}
```

5. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

In the web interface, this service should now be visible in the **Services** section.

| Host ↕            | Service ↕ | Status ↕ | Last Check ↕ | Duration ↕   | Attempt ↕ | Status Information                        |
|-------------------|-----------|----------|--------------|--------------|-----------|-------------------------------------------|
| crete.example.net | TRAP      | PENDING  | N/A          | 0d 0h 0m 5s+ | 1/1       | Service is not scheduled to be checked... |

- Verify that the `submit_check_result` script actually works by invoking it with a test string on the monitoring server:

```
# /usr/local/nagios/libexec/eventhandlers/submit_check_result
crete.example.net TRAP 0 'Everything working'
```

After a short delay, if this has worked correctly, we should see the service change its state in the web interface to reflect the test.

| Host ↕            | Service ↕ | Status ↕ | Last Check ↕        | Duration ↕  | Attempt ↕ | Status Information |
|-------------------|-----------|----------|---------------------|-------------|-----------|--------------------|
| crete.example.net | TRAP      | OK       | 11-25-2015 15:57:45 | 0d 0h 0m 7s | 1/1       | Everything working |

We now need to configure `snmptrapd` and `snmpd` to receive traps and call the `submit_check_result` script for us. Follow the given steps to do so:

- Configure `snmpd` to pass the received traps to `snmptt` by changing its configuration file `/etc/snmp/snmptrapd.conf`. The following configuration may work:

```
traphandle default /usr/sbin/snmptt
disableAuthorization yes
donotlogtraps yes
```

- Restart `snmpd` to apply this change:

```
# /etc/init.d/snmpd restart
```

- Configure `snmptt` to convert IP addresses to hostnames by changing the value for `dns-enable` to 1 in `/etc/snmp/snmptt.ini`:

```
dns_enable = 1
```

- Configure `snmptt` to use Net-SNMP at startup in `/etc/snmp/snmptt.ini`:

```
net_snmp_perl_enable = 1
```

- Configure `snmptt` to respond to an SNMP event by defining it in `/etc/snmp/snmptt.conf`. Here, we've used the generic `linkDown` event defined by the OID `.1.3.6.1.6.3.1.1.5.3` by adding a line to the already existing block:

```
EVENT linkDown .1.3.6.1.6.3.1.1.5.3 "Status Events" Normal
FORMAT Link down on interface $1. Admin state: $2. Operational state:
$3
EXEC /usr/local/nagios/libexec/eventhandlers/submit_check_result $r
```

**TRAP 1 "linkDown for interface \$1"**  
**SDESC**

6. A linkDown trap signifies that the SNMP entity, acting in an agent role, has detected that the ifOperStatus object for one of its communication links is about to enter the down state from some other state (but not from the notPresent state). This other state is indicated by the included value of ifOperStatus.

## EDESC

Depending on your distribution, there may already be a definition for a linkDown event, in which case you may only need to change the EXEC field.

7. Restart snmptt to apply these changes:

```
# /etc/init.d/snmptt restart
```

8. From the monitored host, fire a test trap for a linkDown event. Substitute olympus.example.net for the name or IP address of your monitoring host. This will require the snmptrap utility to be installed on that host and may require root privileges:

```
# snmptrap -v 1 -c public olympus.example.net .1.3.6.1.6.3.1.1.5.3  
localhost 2 0 '' .1.3.6.1.2.1.2.2.1.1.1 i 1
```

## Tip

Note that we use the public community string here; your own will likely differ.

9. Check the Nagios Core log file at /usr/local/nagios/var/nagios.log to check whether there's a new output from the event handler:

```
[1448422541] EXTERNAL COMMAND:  
PROCESS_SERVICE_CHECK_RESULT;crete.example.net;TRAP;1;linkDown for  
interface 1  
[1448422541] PASSIVE SERVICE CHECK: crete.example.net;TRAP;1;linkDown  
for interface 1  
[1448422541] SERVICE ALERT:  
crete.example.net;TRAP;WARNING;HARD;1;linkDown for interface 1  
[1448422541] SERVICE NOTIFICATION:  
nagiosadmin;crete.example.net;TRAP;WARNING;notify-service-by-  
email;linkDown for interface 1
```

If there's a new output, the same state should be reflected for the TRAP service in the web interface.

| Host              | Service | Status | Last Check          | Duration     | Attempt | Status Information       |
|-------------------|---------|--------|---------------------|--------------|---------|--------------------------|
| crete.example.net | TRAP    | ?      | 11-25-2015 16:35:41 | 0d 0h 0m 16s | 1/1     | linkDown for interface 1 |

With this done, we've confirmed that SNMP traps from the crete.example.net host can be received and processed by the olympus.example.net server. We can apply the same setup for other hosts that generate SNMP traps in our network by configuring them to send

their traps to the Nagios Core monitoring server and adding appropriate handlers for the expected traps.

If this doesn't work, the first thing to check should be that your monitoring server is actually listening for checks on the relevant IP address, as `snmptrap` does not throw errors when it cannot deliver traps. On Debian-derived systems, you should verify that the `snmptrapd` process is actually running; it may require a change to `/etc/default/snmptrapd` and a restart of `snmptrapd`.

## How it works...

When an SNMP trap is generated and delivered to the monitoring server by whatever means, the `snmpd` daemon passes it to the `snmptt` program for processing.

The `snmptt` handler checks whether the event OID matches any of the traps for which it has defined events in `snmptt.conf`. In our example, the program finds a handler defined for the OID `.1.3.6.1.6.3.1.1.5.3`, which corresponds to the `linkDown` event, with the number of relevant interfaces as an additional argument in `$1`.

Using this information, the program fires the `submit_check_result` handler that we installed in the first part of the recipe, setting the state of the TRAP service to `WARNING` and including the information **linkDown for interface 1**, as specified by the final argument to `submit_check_result` in the `EXEC` handler. The service can be set to notify the appropriate contacts or contact groups, just as it would for an actively monitored service.

If a trap arrives on the Nagios Core server for a host that Nagios Core doesn't know about, even if it has an event handler defined for `snmptt`, Nagios Core will simply ignore it.



# There's more...

In order to *clear* the state of the service and return it to OK, we can simply schedule an active check for it from the web interface with the **Force Check** tick.

**Service Commands**

- ✓ Enable active checks of this service
- 🕒 Re-schedule the next check of this service
- ? Submit passive check result for this service
- ✗ Stop accepting passive checks for this service
- ✗ Stop obsessing over this service
- ✓ Acknowledge this service problem
- ✗ Disable notifications for this service
- 🕒 Delay next service notification
- 📧 Send custom service notification
- 🕒 Schedule downtime for this service
- ✗ Disable event handler for this service
- ✗ Disable flap detection for this service

**Command Options**

Host Name:

Service:

Check Time:

Force Check: ☒

Because check\_command is defined as check-host-alive, as long as the monitoring host is actually responding to PING, the service should assume the OK state:

| Host ↕            | Service ↕ | Status ↕ | Last Check ↕        | Duration ↕   | Attempt ↕ | Status Information                        |
|-------------------|-----------|----------|---------------------|--------------|-----------|-------------------------------------------|
| crete.example.net | TRAP ?    | OK       | 11-25-2015 16:40:49 | 0d 0h 0m 13s | 1/1       | PING OK - Packet loss = 0%, RTA = 0.23 ms |

## See also

- *Monitoring the output of an SNMP query*, [Chapter 5](#), *Monitoring Methods*
- *Creating an SNMP OID to monitor*, [Chapter 5](#), *Monitoring Methods*
- The *Allowing and submitting passive checks* recipe in this chapter
- The *Submitting passive checks from a remote host with NSCA* recipe in this chapter
- The *Setting up an event handler script* recipe in this chapter



# Setting up an event handler script

In this recipe, you'll learn how to set up an event handler script for Nagios Core. Event handlers are commands that are run on every state change for a host or service (whether for all hosts or services or just particular ones). They are defined in a similar way to notification commands and check commands for plugins.

In this example, we'll implement a simple event handler that writes the date, the host state, and the number of check attempts to a separate file for a single host. This is a simple example to demonstrate the concept; a more practical and complex application for the use of event handlers is given in the *Setting up a redundant monitoring host* recipe in [Chapter 10](#), *Security and Performance*.

# Getting ready

You will need a server running a Nagios Core 4.0 or higher version. You should be familiar with defining new commands, as per the *Creating a new command* recipe in [Chapter 2](#), *Working with Commands and Plugins*, and the *Defining a custom notification method* recipe in [Chapter 4](#), *Configuring Notifications*.

# How to do it...

We can set up a new event handler for the Nagios Core server as follows:

1. Change to the objects configuration directory for Nagios Core. In the quick start guide installation, this directory is `/usr/local/nagios/etc/objects`. Edit the `commands.cfg` file:

```
# cd /usr/local/nagios/etc/objects
# vi commands.cfg
```

2. Add the following command definition:

```
define command {
    command_name    record_host_data
    command_line    /usr/bin/printf "%b" "$LONGDATETIME$: $HOSTSTATE$
(attempt $HOSTATTEMPT$)\n" >>/usr/local/nagios/var/states-
$HOSTNAME$.log
}
```

3. Edit the file containing the definition for an existing host; in this example, `delphi.example.net`. Add the `event_handler` directive with the `record_host_data` value to your host definition:

```
define host {
    use                linux-server
    host_name          delphi.example.net
    alias              delphi
    address            192.0.2.26
    event_handler      record_host_data
}
```

4. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

With this done, the next time the host changes state (whether a SOFT or HARD state, it should log information in the file `/usr/local/nagios/var/states-delphi.example.net.log`:

```
Wed Nov 25 16:44:46 NZDT 2015: DOWN (attempt 1)
Wed Nov 25 16:44:46 NZDT 2015: DOWN (attempt 2)
Wed Nov 25 16:44:46 NZDT 2015: DOWN (attempt 3)
Wed Nov 25 16:44:46 NZDT 2015: UP (attempt 1)
```

# How it works...

The `event_handler` command that we defined is configured to use `printf` to write a line of text to a file named after the host. Its definition is built out of the following four macros:

- `$LONGDATETIME$`: This refers to the date and time in a human-readable format
- `$HOSTSTATE$`: This refers to the state of the host (UP, DOWN, or UNREACHABLE)
- `$HOSTATTEMPT$`: This refers to the number of check attempts made so far for a host in a problem state
- `$HOSTNAME$`: This refers to the hostname itself (used to build the name of the file)

Note that this behavior is slightly different from notifications. Notification commands are only run to alert somebody about the problem when the number of `max_check_attempts` for a host or service have been exceeded. Event handlers are run on `SOFT` changes as well as `HARD` changes and hence can be used to keep more information about the host's performance that might be missed by routine notifications.

Service event handlers can be defined in just the same way, by adding the `event_handler` directive to their definitions:

```
define service {
    use                generic-service
    host_name          crete.example.net
    service_description PING
    check_command       check_ping!100,10%!200,20%
    event_handler      record_service_data
}
```

In this case, we would probably want to use the macros for service states instead:

```
define command {
    command_name       record_service_data
    command_line        /usr/bin/printf "%b" "$LONGDATETIME$: $SERVICESTATE$
(attempt $SERVICEATTEMPT$)\n" >>/usr/local/nagios/var/states-$HOSTNAME$-
$SERVICEDESC$.log
}
```

## There's more...

As a shortcut, if there's an event handler we want to run on all hosts or all services, we can use the `global_host_event_handler` and `global_service_event_handler` directives in `nagios.cfg`:

```
global_host_event_handler=record_host_data
global_service_event_handler=record_service_data
```

This will apply the appropriate event handlers to all hosts and services, and the handlers will therefore run whenever a host or service changes state.

A specialized case of using an event handler to record detailed performance data of plugins and checks is also possible using Nagios Core's Performance Data feature, as documented in the manual, which can be found at

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/perfddata.html>.

Performance data is written on every check rather than every state change and is hence useful to assess the performance of plugins and checks. Performance data is used by the Nagiosgraph utility, as discussed in the *Tracking host and service states with Nagiosgraph* recipe in this chapter.



## See also

- *Creating a new command*, [Chapter 2](#), *Working with Commands and Plugins*
- *Defining a custom notification method*, [Chapter 4](#), *Configuring Notifications*
- *The Tracking host and service states with Nagiosgraph* recipe in this chapter



# Tracking host and service states with Nagiosgraph

In this recipe, you'll learn how to install and configure **Nagiosgraph**, which is a program that is integrated with Nagios Core's performance data tools to produce graphs that show long-term information about how checks for hosts and services are performing.

# Getting ready

You will need to run a Nagios Core 4.0 or later server. Nagiosgraph will probably still work with older versions of Nagios Core, but the configuration may be slightly different. The `INSTALL` document included in the source for Nagiosgraph explains the differences in detail.

You should have a thorough understanding of defining hosts, services, and commands, and you should be able to install new software as the root user on the monitoring server. You should also be, at least, familiar with the layout of your Apache HTTPD server on the monitoring system; this recipe will assume that it is installed in `/usr/local/apache`.

Because Nagiosgraph has many Perl dependencies, you will need to have Perl installed on your server and you will likely also need to install a few Perl modules as dependencies. The package manager for your system may include them or you may need to download them using the Comprehensive Perl Archive Network (CPAN), which you can find at <http://www.cpan.org/modules/INSTALL.html>.

The server will need to already be monitoring at least one host with at least one service for the graphs to be of any use. Nagiosgraph includes rule sets that translate known performance data strings into usable statistics. This means that graphing will work well for familiar plugins with a predictable output format, such as `check_ping` or `check_http`, but might not graph data for less commonly used plugins without a little custom configuration.

This recipe is not a comprehensive survey of everything you can do with Nagiosgraph; if you like what this does, make sure to check out Nagiosgraph's documentation online at <http://nagiosgraph.sourceforge.net/>.

# How to do it...

We can get some basic Nagiosgraph functionality for our monitoring server by following these steps:

1. Download the latest version of Nagiosgraph from its website at <http://nagiosgraph.sourceforge.net/> directly onto your monitoring server using a tool such as wget:

```
$ cd
$ wget
http://downloads.sourceforge.net/project/nagiosgraph/nagiosgraph/1.5.2/
nagiosgraph-1.5.2.tar.gz
```

2. Inflate the .tar.gz file and change into the uncompressed directory:

```
$ tar xzf nagiosgraph-1.5.2.tar.gz
$ cd nagiosgraph-1.5.2
```

3. As the root user, run the install.pl script with the --check-prereq option. This will give you a survey of any dependencies you may need to install via Debian packages or CPAN. When you have installed all the prerequisites, the output should look something like this:

```
# ./install.pl --check-prereq
checking required PERL modules
Carp...1.3301
CGI...4.09
Data::Dumper...2.151_01
Digest::MD5...2.53
File::Basename...2.85
File::Find...1.27
MIME::Base64...3.14
POSIX...1.38_03
RRDs...1.4008
Time::HiRes...1.9726
checking optional PERL modules
GD...2.53
Nagios::Config... ***FAIL***
checking nagios installation
found nagios executable at /usr/local/nagios/bin/nagios
found nagios init script at /etc/init.d/nagios
checking web server installation
found apache executable at /usr/sbin/apache2
found apache init script at /etc/init.d/apache2
```

These are all reasonably standard Perl libraries, so don't forget to check whether the packages are available for them before you resort to using CPAN. For example, I was able to install the RRDs and GD modules on my Debian system as follows:

```
# apt-get install librrds-perl libgd-gd2-perl
```

If you are having trouble getting install.pl to find your Nagios Core or Apache HTTPD instances, take a look at the output of install.pl --help to run an

installation specific to your kind of system. This is documented in more detail in the INSTALL file.

4. As the root user, run the `install.pl` script with the `--install` argument. You will be prompted many times for directory layout options. The default is shown in square brackets and should be correct for a typical Nagios Core installation, so, to start with, simply press *Enter* on each option:

```
# ./install.pl --install
...Destination directory (prefix)? [/usr/local/nagiosgraph]
Location of configuration files (etc-dir)? [/usr/local/nagiosgraph/etc]
Location of executables? [/usr/local/nagiosgraph/bin]
Location of CGI scripts? [/usr/local/nagiosgraph/cgi]
Location of documentation (doc-dir)? [/usr/local/nagiosgraph/doc]
Location of examples? [/usr/local/nagiosgraph/examples]
Location of CSS and JavaScript files? [/usr/local/nagiosgraph/share]
Location of utilities? [/usr/local/nagiosgraph/util]
Location of state files (var-dir)? [/usr/local/nagiosgraph/var]
Location of RRD files? [/usr/local/nagiosgraph/var/rrd]
Location of log files (log-dir)? [/usr/local/nagiosgraph/var]
Path of log file? [/usr/local/nagiosgraph/var/nagiosgraph.log]
Path of CGI log file? [/usr/local/nagiosgraph/var/nagiosgraph-cgi.log]
URL of CGI scripts? [/nagiosgraph/cgi-bin]
URL of CSS file? [/nagiosgraph/nagiosgraph.css]
URL of JavaScript file? [/nagiosgraph/nagiosgraph.js]
Path of Nagios performance data file? [/tmp/perfdata.log]
URL of Nagios CGI scripts? [/nagios/cgi-bin]
username or userid of Nagios user? [nagios]
username or userid of web server user? [www-data]
Modify the Nagios configuration? [n]
Modify the Apache configuration? [n]
...
```

After the preceding selections are all made, the files should be installed with appropriate permissions set. The final part of the output gives instructions to add configuration to Nagios Core and Apache HTTPD, which we'll do next.

5. Change to the Nagios Core configuration directory. In the quick start guide installation, this directory is `/usr/local/nagios/etc`:

```
# cd /usr/local/nagios/etc
```

6. Edit the core configuration file `nagios.cfg`, and add the following directives at the end of the file:

```
# process nagios performance data using nagiosgraph
process_performance_data=1
service_perfdata_file=/tmp/perfdata.log
service_perfdata_file_template=$LASTSERVICECHECK$|$HOSTNAME$|$SERVICE
DESC$|$SERVICEOUTPUT$|$SERVICEPERFDATA$
service_perfdata_file_mode=a
service_perfdata_file_processing_interval=30
service_perfdata_file_processing_command=process-service-perfdata-for-
nagiosgraph
```

7. Change to the Nagios Core objects configuration directory. In the quick start guide installation, this directory is `/usr/local/nagios/etc/objects`:

```
# cd /usr/local/nagios/etc/objects
```

8. Edit the `commands.cfg` file and add the following command definition:

```
# command to process nagios performance data for nagiosgraph
define command {
    command_name process-service-perfdata-for-nagiosgraph
    command_line /usr/local/nagiosgraph/bin/insert.pl
}
```

9. Edit the `httpd.conf` file for your Apache HTTPD server to include the following line at the end:

```
Include /usr/local/nagiosgraph/etc/nagiosgraph-apache.conf
```

In a local install of Apache HTTPD, this file is normally in `/usr/local/apache/conf/httpd.conf`, but its location varies as per your system. On Debian-derived systems, it may be `/etc/apache2/apache2.conf`.

10. Validate the configuration of both the Apache HTTPD server and the Nagios Core server and restart them both:

```
# /usr/local/apache/bin/apachectl configtest
# /usr/local/apache/bin/apachectl restart
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

11. Visit `http://olympus.example.net/nagiosgraph/cgi-bin/showconfig.cgi` in your browser, substituting your own Nagios Core server's hostname to check whether everything's working. You should see a long page with configuration information for Nagiosgraph after this:

**nagiosgraph configuration on olympus.example.net**  
25 Nov 2015 16:58:21 NZDT

**PERL modules**

required

☐

Carp: 1.3301

☐

☐

CGI: 4.09

☐

☐

Data::Dumper: 2.151\_01

☐

☐

Digest::MD5: 2.53

☐

☐

File::Basename: 2.85

☐

☐

File::Find: 1.27

☐

☐

MIME::Base64: 3.14

☐

☐

POSIX: 1.38\_03

☐

☐

RRDs: 1.4008

☐

☐

Time::HiRes: 1.9726

☐

optional

☐

GD: 2.53

☐

☒

Nagios::Config: not installed

☒

**nagiosgraph**

☐

ngshared.pm: ok

☐

☐

version: 1.5.2

☐

☐

nagiosgraph.conf: ok

☐

☐

RRD directory: ok

☐

☐

log file: ok

☐

☐

CGI log file: ok

☐

☐

map file: ok

☐

**Environment**

```

CONTEXT_DOCUMENT_ROOT=/usr/local/nagiosgraph/cgi
CONTEXT_PREFIX=/nagiosgraph/cgi-bin
DOCUMENT_ROOT=/var/www/html
GATEWAY_INTERFACE=CGI/1.1
HTTP_ACCEPT=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
HTTP_ACCEPT_ENCODING=gzip, deflate

```

12. If everything works well up to this point, the only thing left to do is define an action URL for the services that you want to graph so that you can click to go directly to the graphs for that service from the Nagios Core web interface.

The tidiest and most straightforward way to do this is to define a service template:

```

define service {
    name          nagiosgraph
    action_url    /nagiosgraph/cgi-bin/show.cgi?
host=$HOSTNAME$&service=$SERVICEDESC$
    register      0
}

```

Then, you can have the services you want graphed inherit from the template, as well as from any other templates they use, by adding a comma and nagiosgraph to the value for the use directive:

```

define service {

```



```

use generic-service,nagiosgraph
host_name corinth.example.net
service_description PING
check_command check_ping!100,10%!200,20%
}

```

You should do this for all the services for which you want graphing.

13. Validate the configuration and restart the Nagios Core server again:

```

# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart

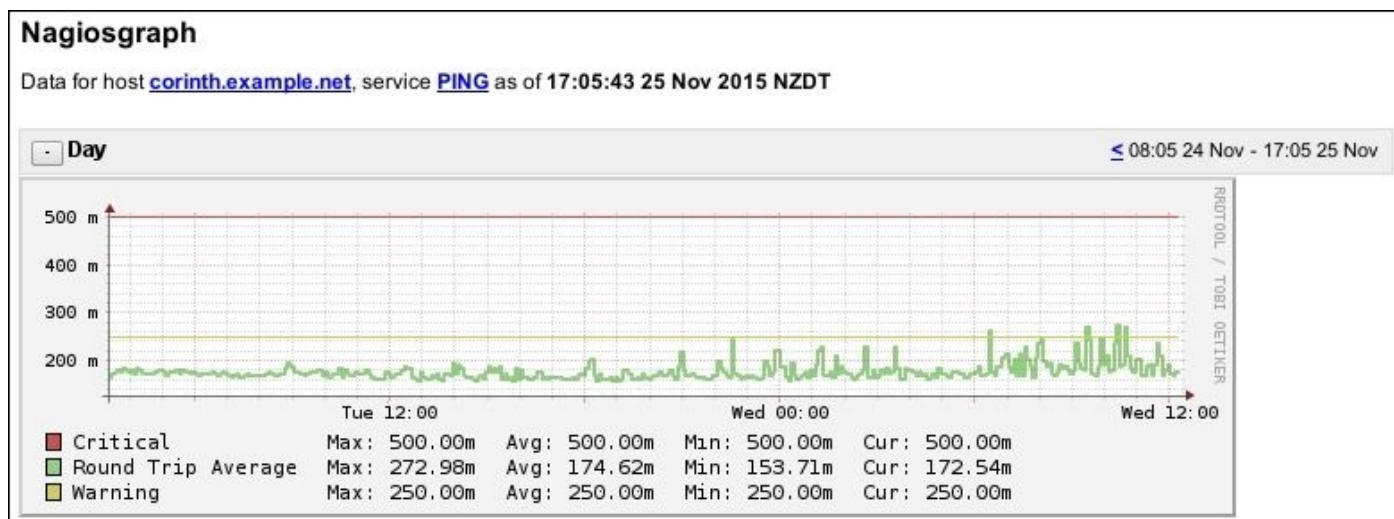
```

With this done, visiting the **Services** section of the web interface should include action icons after each graphed service:

| Host ↕            | Service ↕                                                                              | Status ↕ | Last Check ↕        | Duration ↕    | Attempt ↕ |
|-------------------|----------------------------------------------------------------------------------------|----------|---------------------|---------------|-----------|
| crete.example.net | BACKUP ?                                                                               | PENDING  | N/A                 | 0d 0h 0m 2s+  | 1/3       |
|                   | HTTP  | OK       | 11-25-2015 17:04:16 | 0d 0h 5m 26s  | 1/3       |
|                   | PING  | OK       | 11-25-2015 17:04:21 | 0d 0h 5m 21s  | 1/3       |
|                   | TRAP  | OK       | 11-25-2015 16:40:49 | 0d 0h 28m 53s | 1/1       |

**Perform Extra Service Actions**

Clicking on one of these should bring up a graph interface; for example, a service using check\_ping might show something like the following:



Note that the graph includes two line bars to show the thresholds for the CRITICAL and WARNING state as well as the actual response time. Also note that the preceding graph is several hours old; it will take a while to build up enough data to see a perceptible line and you may not see any graphs until performance data has actually been received from Nagios by Nagiosgraph.

Don't forget that graphs won't work out of the box for every service. If Nagiosgraph doesn't know how to parse the performance data for a check, it will show red error text

instead of graphs. We'll mention an approach to fix this in the *There's more* section.

## How it works...

The preceding configuration changes prompt Nagios Core to log performance data for every check on every service using the `service_perfdata_file_processing_command` directive. This command, named `process-service-perfdata-for-nagiosgraph`, is defined to pass data to the `bin/insert.pl` script included in the new `/usr/local/nagiosgraph` directory.

This script, in turn, parses performance output, such as the following output from a typical service using `check_ping`:

```
PING OK - Packet loss = 0%, RTA = 174.19 ms
```

Nagiosgraph extracts numeric information from the performance data, according to the templates defined in `/usr/local/nagiosgraph/etc/map`, using Perl's regular expressions. This data is recorded using Perl's bindings for the RRD library and is graphed using the GD2 library with an appearance similar to graphs produced by MRTG.

The `action_url` directive uses macros for each service to define a URL for each service that shows its graphs. In our example, for a service PING on the `corinth.example.net` host, the `action_url` command would expand to the following:

```
action_url  /nagiosgraph/cgi-bin/show.cgi?  
host=corinth.example.net&service=PING
```

This isn't the only possible use of `action_url`, of course; it just happens to be a useful one in our case. You can make the `action_url` command go anywhere you'd like for a given host or service.

## There's more...

You may well be running some kind of check that Nagiosgraph isn't able to graph because it doesn't understand the format of the performance output and can't extract numeric information from it. The default mapping rules cover output from quite a few standard plugins, but if you know a little Perl, you should be able to add more rules to `/usr/local/nagiosgraph/etc/map` to process other kinds of plugin output.

Besides the examples already in the map file, which includes instructions for writing new output checks, there are more examples of such definitions included in the `/usr/local/nagiosgraph/examples/map_examples` file.

If you're comparing Nagios graphing solutions, another popular solution to try could be PNP4Nagios, which is available at <http://docs.pnp4nagios.org/pnp-0.6/start>.

## See also

- *Monitoring Nagios performance with Nagiosstats*, [Chapter 10](#), *Security and Performance*
- The *Getting extra visualizations with NagVis* recipe in this chapter



# Reading status in a MySQL database with NDOUtils

In this recipe, you'll learn how to install the NDOUtils extension to Nagios Core in order to have all of Nagios Core's configuration and data written into a MySQL database. This allows us to easily develop custom reports and interfaces for Nagios Core data with languages such as Perl and PHP and their standard interfaces for the popular MySQL server, as opposed to interacting with Nagios Core's own logs or data format. Some plugins such as NagVis use this format to read information about Nagios Core configuration and objects.

# Getting ready

You will need a Nagios Core server version 4.0 or later. NDOUtils will probably still get installed and work on older versions of Nagios Core, but the installation process is slightly different; refer to the `INSTALL` file included in the NDO source for more information on this.

Nagios Core uses its event broker functionality to write information to the socket for the MySQL database to pick up. You will need to compile Nagios Core with the `--enable-event-broker` flag:

```
$ ./configure --enable-event-broker
$ make
# make install
```

If you are unsure whether you've compiled Nagios Core with this flag, it is probably a good idea to recompile and reinstall Nagios Core from your original sources with this installed. Don't forget to back up your previous installation in case of problems.

In order to compile the `ndomod` part of NDOUtils, you will need to have the MySQL client libraries and headers installed on the Nagios Core server. You will also need to have a MySQL server ready to store the data. You don't need to run the MySQL server on the same host as Nagios Core, but the Nagios Core server should be able to connect to it.

Finally, you should take note of the opening paragraph of `README`, which at the time of writing points out that NDOUtils for Nagios Core 4.0 is still officially in beta; you should read the note and be aware of the risks in installing it. In the author's own experience, however, the code is stable.



# How to do it...

We can install the NDOUtils package for Nagios Core as follows:

1. Download the latest NDOUtils .tar.gz from its Sourceforge site at <http://sourceforge.net/projects/nagios/files/ndoutils-2.x/>:

```
$ wget http://downloads.sourceforge.net/project/nagios/ndoutils-2.x/ndoutils-2.0.0/ndoutils-2.0.0.tar.gz
```

2. Unpack .tar.gz and change into the uncompressed directory:

```
$ tar xzf ndoutils-2.0.0.tar.gz
$ cd ndoutils-2.0.0
```

3. Run ./configure and make to build the software. Note that there is no install target; we will be performing the installation manually:

```
$ ./configure
$ make
```

Read the output of ./configure carefully if the build fails, to determine if you are missing any dependencies on your system. The output of ./configure should end with something like this:

```
*** Configuration summary for ndoutils 2.0.0 02-28-2014 ***:
```

```
General Options:
```

```
-----
```

```
ND02DB user:      nagios
```

```
ND02DB group:     nagios
```

```
Review the options above for accuracy. If they look okay,
type 'make' to compile the NDO utilities.
```

4. On the MySQL server, create a database to store the Nagios Core information and a user account to access it. In this example, the MySQL server is running on the same host as the Nagios Core server (olympus.example.net), so the access will be done from localhost:

```
mysql> CREATE DATABASE nagios;
Query OK, 1 row affected (0.10 sec)
mysql> CREATE USER 'ndo'@'localhost' IDENTIFIED BY 'mPYxbAYqa';
Query OK, 0 rows affected (0.62 sec)
mysql> GRANT ALL ON nagios.* TO 'ndo'@'localhost';
Query OK, 0 rows affected (0.02 sec)
```

We have used a random password after IDENTIFIED BY. You should generate your own secure password.

5. Run the installdb script in the source to create the various tables Nagios Core will use. Use the database details established in the previous step:

```
$ cd db
$ ./installdb -u ndo -p mPYxbAYqa -h localhost -d nagios
** Creating tables for version 2.0.1
```

**Using mysql.sql for installation...**

**\*\* Updating table nagios\_dbversion**

**Done!**

Don't be concerned about the following error message; it appears because you are installing the extension for the first time:

**DBD::mysql::db do failed: Table 'nagios.nagios\_dbversion' doesn't exist at ./installdb line 51.**

6. Copy the compiled ndomod-4x.o module into the /usr/local/nagios/bin directory:

```
# cd ..  
# cp src/ndomod-4x.o /usr/local/nagios/bin/ndomod.o
```

7. Copy the sample configuration for the module into /usr/local/nagios/etc:

```
# cp config/ndomod.cfg-sample /usr/local/nagios/etc/ndomod.cfg
```

8. Make sure that it is readable only by the nagios user:

```
# chown nagios.nagios /usr/local/nagios/etc/ndomod.cfg  
# chmod 0600 /usr/local/nagios/etc/ndomod.cfg
```

9. Edit your nagios.cfg file:

```
# vi /usr/local/nagios/etc/nagios.cfg
```

10. Add a broker\_module definition to the file and verify that the event\_broker\_options directive is set to -1:

```
broker_module=/usr/local/nagios/bin/ndomod.o  
config_file=/usr/local/nagios/etc/ndomod.cfg  
event_broker_options=-1
```

Note that the broker\_module and config\_file definitions should be on the same line, but event\_broker\_options should be on a separate line.

With this done, the broker module is said to be successfully installed and we can move on to installing the ndo2db daemon. To install the same, follow these steps:

1. Copy the ndo2db-4x binary into the /usr/local/nagios/bin directory:

```
# cp src/ndo2db-4x /usr/local/nagios/bin/ndo2db
```

2. Copy the sample configuration for the daemon into /usr/local/nagios/etc:

```
# cp config/ndo2db.cfg-sample /usr/local/nagios/etc/ndo2db.cfg
```

3. Make sure that it is readable only by the nagios user:

```
# chown nagios.nagios /usr/local/nagios/etc/ndo2db.cfg  
# chmod 0600 /usr/local/nagios/etc/ndo2db.cfg
```

4. Edit the configuration file as installed:

```
# vi /usr/local/nagios/etc/ndo2db.cfg
```

5. Change the values in `ndo2db.cfg` to reflect the database details:

```
db_host=localhost
db_port=3306
db_name=nagios
db_user=ndo
db_pass=mPYxbAYqa
```

6. Test the `ndo2db` daemon by starting it and verifying that it is running with `pgrep(1)`:

```
# /usr/local/nagios/bin/ndo2db -c /usr/local/nagios/etc/ndo2db.cfg
# pgrep ndo2db
32285
```

If it works, you should add this command into your system's `init` scripts so that the daemon is started at boot time. The `daemon-init` file in the distribution may be helpful regarding this.

7. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

8. With this done, inspecting the database tables in MySQL should show that they have been filled with information from Nagios Core, for example, the `nagios_services` table:

```
$ mysql --user=ndo --password --database=nagios
mysql> select count(1) from nagios_services;
+-----+
| count(1) |
+-----+
|      54 |
+-----+
1 row in set (0.00 sec)
```

# How it works...

NDOUtils is in fact a collection of components, two of which we have installed in the previous section:

- `ndomod.o`: This is used as a broker module to write events and data from Nagios Core to a Unix socket in `/usr/local/nagios/var/ndo.sock`. It runs as a module of the Nagios Core server.
- `ndo2db`: This is used as a database backend to read the events and data from the Unix socket to which `ndomod` writes and to apply them as MySQL database operations. It runs independently as a daemon on the system and performs the MySQL connections and transactions.

The `broker_module` command updates these tables as plugins are run, hosts and services change state, notifications are issued, and other Nagios Core behavior takes place. It covers most of the data of interest to us quite comprehensively. Note that it includes the following information:

- Configuration directives
- Details for types of Nagios Core objects
- Properties and current states of hosts
- Acknowledgement and scheduled downtime information
- Notification history and complete logging

The main reason to install NDOUtils is to put Nagios Core's data into a standardized format so that it can be read and processed by external applications, whether simple table-style reports or completely new application interfaces to the Nagios Core data. This tends to be much easier than custom building a Nagios Core CGI of your own!

## There's more...

NDOUtils is useful, but some administrators find the overhead of maintaining the database server for it unnecessary. For very large installations with a lot of configuration and historical data, queries on the data can also be slow or otherwise delayed. This leads many administrators to prefer the use of MK Livestatus as an alternative. This program is also discussed in this chapter in the *Reading status from a UNIX socket with MK Livestatus* recipe.

To get the most out of NDOUtils, it's a good idea to take a look at its documentation at <https://assets.nagios.com/downloads/nagioscore/docs/ndoutils/NDOUtils.pdf>.

## See also

- The *Reading status from a Unix socket with MK Livestatus* recipe in this chapter
- The *Writing customized Nagios Core reports* recipe in this chapter
- The *Getting extra visualizations with NagVis* recipe in this chapter



# Reading status from a Unix socket with MK Livestatus

The NDOUtils event broker and database system is a useful way of getting data out of the Nagios Core program and its history in order to generate our own reports and displays but it has a few flaws; the MySQL database setup can be complicated and a hassle to maintain, and the database can grow quite large, which makes searching slow and uses a lot of system resources not just for data searching, but for the *housekeeping* it does in removing old entries.

The **MK Livestatus** tool, part of Mathias Kettner's Check\_MK suite of Nagios Core monitoring tools, provides a way of retrieving data directly from the running Nagios process by writing queries to a Unix socket and by reading results back from it, without requiring a database to do so. Provided our queries do not need to be too sophisticated, this extension can make things a lot faster and less burdensome on system resources, which is why a growing number of administrators and Nagios plugin developers prefer this to the older NDOUtils.

In this recipe, we'll install MK Livestatus as an event broker in a similar fashion to how we installed NDOUtils in the *Reading status into a MySQL database with NDOUtils* recipe in this chapter. The installation is somewhat simpler than NDOUtils's installation.



# Getting ready

You'll need a server running Nagios Core 4.0 or greater version with at least a few hosts and services configured. Unlike NDOUtils, you do not need to have a MySQL server running. If you already have Check\_MK installed from the *Using check\_mk instead of NRPE* recipe in [Chapter 6, Enabling Remote Execution](#), you may already have MK Livestatus installed; look for a line similar to this in your `nagios.cfg` file:

```
broker_module=/usr/local/nagios/lib/mk-livestatus/livestatus.o  
/usr/local/nagios/var/rw/live
```

# How to do it...

We can install MK Livestatus on our Nagios Core server by following these steps:

1. Download the most recent version of the check\_mk suite's source; at the time of writing, I downloaded it from [http://mathias-kettner.com/check\\_mk\\_download\\_version.php](http://mathias-kettner.com/check_mk_download_version.php). It is a large file (100MB):

```
$ wget https://mathias-kettner.de/support/1.2.6p9/check-mk-raw-1.2.6p9.cre.tar.gz
```

2. Unpack the downloaded tarball and move it into the packages/mk-livestatus directory within it:

```
$ tar xzf check-mk-raw-1.2.6p9.cre.tar.gz
$ cd check-mk-raw-1.2.6p9packages/mk-livestatus
```

3. Unpack the mk-livestatus-1.2.6p9.tar.gz file in that directory and move it into the directory we created:

```
$ tar xzf mk-livestatus-1.2.6p9.tar.gz
$ mv mk-livestatus-1.2.6p9
```

4. Run the configure script with the --with-nagios4 option, set --prefix to the Nagios Core installation directory, and check whether it succeeds; then, run make and then run make install as root once the build completes:

```
$ ./configure --prefix=/usr/local/nagios --with-nagios4
$ make
# make install
```

5. Add the following lines to the end of /usr/local/nagios/etc/nagios.cfg:

```
broker_module=/usr/local/nagios/lib/mk-livestatus/livestatus.o
/usr/local/nagios/var/rw/live
event_broker_options=-1
```

6. Validate the configuration and restart the Nagios Core server:

```
# /usr/local/nagios/bin/nagios -v /usr/local/nagios/etc/nagios.cfg
# /etc/init.d/nagios restart
```

Having done this, we should now be able to retrieve and test data from the running Nagios Core server using the unixcat tool included in the distribution; in this example, we're retrieving the hostname and state index:

```
$ printf '%s\n' 'GET hosts' 'Columns: name state' |
/usr/local/nagios/bin/unixcat /usr/local/nagios/var/rw/live
athens.example.net;0
crete.example.net;0
sparta.example.net;0
```

## How it works...

The MK Livestatus suite creates a `/usr/local/nagios/lib/mk-livestatus` directory and writes an event broker into it named `livestatus.o`. We configure `nagios.cfg` to use this module as its event broker in the same way as NDOUtils, with the `broker_module` directive pointing to both the paths to the binary and pointing to the path to the Unix socket that it should create, `/usr/local/nagios/var/rw/live`.

This socket is the communication point to write queries for MK Livestatus. The answers to these queries will be provided with the data from the running Nagios Core process; most programming languages will have a good language or library support to write data to Unix sockets, so this should allow us to retrieve data in whatever environment or application we need to, including Perl, Python, Ruby, or shell scripts.

We wrote a simple query of two lines in the socket, in the **Livestatus Query Language (LQL)**, retrieving two columns of information for all our configured hosts:

```
GET hosts
Columns: name status
```

Because the data is retrieved directly from the running Nagios Core process, rather than from a database, most queries are answered very quickly.

## There's more...

Most of the information available in NDOUtils can be retrieved using LQL queries. Some example queries that you can try are as follows:

```
GET contacts  
Columns: name alias
```

This query retrieves the values of the name and alias attributes for all of the configured contacts:

```
GET services  
Columns: host_name description  
Filter: state != 0
```

This retrieves the host\_name and service\_description attributes for all the configured services that are not presently in an OK state:

```
GET services  
Columns: host_name description  
OutputFormat: json  
Filter: state != 0
```

This query does the same as the preceding query, but prints its output in a JSON format, which may be easier for some programming language libraries to parse than the usual CSV output.

Complete documentation on the features and syntax of LQL is available on the Check\_MK website at [https://mathias-kettner.de/checkmk\\_livestatus.html](https://mathias-kettner.de/checkmk_livestatus.html).

## See also

- The *Reading status from a Unix socket with NDOUtils* recipe in this chapter
- Using *check\_mk* instead of *NRPE*, [Chapter 6](#), *Enabling Remote Execution*
- The *Writing customized Nagios Core reports* recipe in this chapter
- The *Getting extra visualizations with NagVis* recipe in this chapter



# Writing customized Nagios Core reports

In this recipe, we'll explore some simple applications of the NDOUtils database by trying out some queries and changing one of them into both a simple daily e-mail in Perl and also into a PHP-based RSS feed.

## Getting ready

This recipe assumes that you have NDOUtils already installed and that your Nagios Core 4.0 or higher server is monitoring at least a few hosts and services so that the queries we try actually return some data. You should also have some means of executing MySQL queries on the database server. The `mysql(1)` command-line client will work just fine; a web-based tool such as phpMyAdmin might make the data a little easier to explore.



# How to do it...

We can explore some queries against the NDOUtils databases as follows:

1. Retrieve the content and date/time of the latest 10 notifications:

```
mysql> SELECT start_time, long_output FROM nagios_notifications ORDER
BY start_time DESC LIMIT 10;
```

2. Retrieve the content and date/time of the latest 10 host or service comments:

```
mysql> SELECT entry_time, comment_data FROM nagios_commenthistory ORDER
BY entry_time DESC LIMIT 10;
```

3. Count the number of hosts currently in the OK state:

```
mysql> SELECT COUNT(1) FROM nagios_hoststatus WHERE current_state = 0;
```

4. Retrieve the names of all hosts that are currently in scheduled downtime:

```
mysql> SELECT display_name FROM nagios_hosts JOIN nagios_hoststatus
USING (host_object_id) WHERE scheduled_downtime_depth > 0;
```

## Tip

Note that the syntax of this query assumes a MySQL version of at least 5.0.12.

5. Return a list of all hostnames and the number of services associated with them:

```
mysql> SELECT nagios_hosts.display_name, COUNT(service_id) FROM
nagios_hosts LEFT JOIN nagios_services ON nagios_hosts.host_object_id =
nagios_services.host_object_id GROUP BY nagios_hosts.host_object_id;
```

We could implement a Perl script to print the latest 10 notifications using the DBI module as follows:

```
#!/usr/bin/env perl
# Enforce Perl best practices
use strict;
use warnings;
# Import database modules
use DBI;
use DBD::mysql;
# Get connection to database
my $nagios = DBI->connect(
    'dbi:mysql:dbname=nagios;host=localhost',
    'ndo',
    'mPYxbAYqa'
) or die "Could not connect to database";
# Define an SQL query to run
my $query = q{
    SELECT
        notification_id, start_time, name1, long_output
    FROM
        nagios_notifications
    JOIN
```

```

        nagios_objects USING (object_id)
ORDER BY
    start_time DESC
LIMIT
    10
};
# Execute query and retrieve notifications
my $notifications = $nagios->selectall_hashref(
    $query,
    'notification_id'
) or die 'Could not retrieve notifications';
# Print each notification
foreach my $id (keys %{$notifications}) {
    my $notification = $notifications->{$id};
    printf {*STDOUT} "%s - %s: %s\n",
        $notification->{start_time},
        $notification->{name1},
        $notification->{long_output};
}
# Exit successfully
exit 0;

```

After saving the code into a latest-notifications.pl file, we could run it like this:

```

$ chmod +x latest-notifications.pl
$ ./latest-notifications.pl2015-11-25 10:11:27 - blog: PING OK - Packet
loss = 0%, RTA = 160.31 ms2015-11-25 10:48:17 - athens: PING OK - Packet
loss = 0%, RTA = 155.82 ms2015-11-25 14:08:17 - athens: PING WARNING -
Packet loss = 16%, RTA = 171.79 ms2015-11-25 14:13:17 - athens: PING OK -
Packet loss = 0%, RTA = 164.39 ms2015-11-25 10:56:17 - athens: PING
CRITICAL - Packet loss = 28%, RTA = 164.65 ms2015-11-25 10:38:17 - athens:
PING CRITICAL - Packet loss = 28%, RTA = 166.10 ms2015-11-25 10:06:27 -
blog: PING WARNING - Packet loss = 16%, RTA = 163.72 ms2015-11-25 13:39:27
- blog: PING WARNING - Packet loss = 16%, RTA = 163.78 ms2015-11-25
13:44:27 - blog: PING OK - Packet loss = 0%, RTA = 167.10 ms2015-11-25
13:29:27 - blog: PING CRITICAL - Packet loss = 28%, RTA = 159.55 ms

```

Similarly, we could implement a crude RSS feed for notifications using PHP5 with PDO MySQL as follows:

```

<?php
// Get connection to database
$nagios = new PDO(
    'mysql:host=localhost;dbname=nagios',
    'ndo',
    'mPYxbAYqa'
);
// Define an SQL query to run
$query = '
SELECT
    start_time, name1, long_output
FROM
    nagios_notifications
JOIN
    nagios_objects
USING

```

```

        (object_id)
ORDER BY
    start_time DESC
LIMIT
    10
';
// Retrieve all the notifications as objects
$statement = $nagios->prepare($query);
$statement->setFetchMode(PDO::FETCH_OBJ);
$statement->execute();
// Read the notifications into an array
for ($notifications = array(); $notification = $statement->fetch(); ) {
    $notifications[] = $notification;
}
// Send an RSS header rather than an HTML one
header("Content-Type: application/rss+xml; charset=utf-8");
echo '<?xml version="1.0" encoding="UTF-8"?>';
?>
<rss version="2.0">
    <channel>
        <title>
            Latest Nagios Core Notifications
        </title>
        <link>
            http://olympus.example.net/nagios/
        </link>
        <description>
            The ten most recent notifications from Nagios Core
        </description>
        <? foreach ($notifications as $notification): ?>
            <item>
                <description>
                    <?= htmlspecialchars($notification->name1); ?>: <?=
htmlspecialchars($notification->long_output) ?>
                </description>
                <pubDate>
                    <?= htmlspecialchars(date('r', strtotime($notification-
>start_time))) ?>
                </pubDate>
            </item>
        <? endforeach; ?>
    </channel>
</rss>

```

After saving it in a file named `latest-notifications.php`, we could subscribe to the feed it produces in our favorite RSS reader, such as Liferea (<http://liferea.sourceforge.net/>).

|                                                                                   | Date                                                                                            | Headline                |
|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|-------------------------|
|                                                                                   | Today 2:13 PM                                                                                   | Notification for athens |
|  | Today 2:08 PM                                                                                   | Notification for athens |
|  | Today 1:44 PM                                                                                   | Notification for blog   |
|  | Today 1:39 PM                                                                                   | Notification for blog   |
|  | Today 1:29 PM                                                                                   | Notification for blog   |
|  | Today 10:56 AM                                                                                  | Notification for athens |
|  | Today 10:48 AM                                                                                  | Notification for athens |
|  | Today 10:38 AM                                                                                  | Notification for athens |
|  | Notification for athens                                                                         |                         |
|  | flag  bookmark |                         |
| athens: PING OK - Packet loss = 0%, RTA = 164.39 ms                               |                                                                                                 |                         |

## How it works...

The preceding examples are just to get you started with very simple reports; there is a wealth of data available in the NDOUtils database to explore. Here are some other possibilities for reports that could be written:

- A breakdown of all the hosts in your system and their states that is classified in an order according to their name and presented in an HTML table
- A list of all the hosts that have been down more than once in a month
- The percentage of uptime for all hosts

## See also

- The *Reading a status into a MySQL database with NDOUtils* recipe in this chapter
- The *Getting extra visualizations with NagVis* recipe in this chapter



# Getting extra visualizations with NagVis

In this recipe, we'll explore how to go beyond the default network map, discussed in [Chapter 8](#), *Managing Network Layout*, to get a lot of visualization power with the NagVis extension. NagVis can use the MK Livestatus backend to build custom maps in various styles.

NagVis is most likely of interest to you if you're interested in visualizing Nagios data more extensively, particularly if you're having problems with the scalability of the included Nagios Core status map. The default status map works well for smaller networks but can struggle with rendering larger ones in a timely fashion.

A complete survey of NagVis' functions would not be possible in one recipe, but this recipe will walk you through downloading, installing, and configuring the extension to give you a simple **automap** in order to get you started.



# Getting ready

You should have a running Nagios Core server with version 4.0 or later and have the MK Livestatus backend successfully installed. This is discussed in the *Reading status from a Unix socket with MK Livestatus* recipe in this chapter. NDOUtils will also work as an alternative backend but may have performance problems.

In order for the automap to be of much use, you will need a network with at least a few parent-child relationships—refer to the *Creating a network host hierarchy* recipe in [Chapter 8, Managing Network Layout](#), for more details on how this is done.

NagVis includes an installation script that deals quite well with different systems' installations of Nagios Core. However, it still requires certain dependencies, specifically the following ones:

- Apache with mod\_php on the same server as Nagios Core
- PHP 5.6 or newer with these modules: gd, gettext, mbstring, mysql, pdo, session, sqlite, and xml
- The Graphviz graph visualization software with these modules: circo, dot, fdp, neato, and twopi
- SQLite 3

You may need to consult your system's documentation to install all of these dependencies; check your system's package manager, if any. For Ubuntu and other Debian-derived systems, the following packages generally suffice:

```
# apt-get install apache2 libapache2-mod-php5 libgd2-xpm libgd2-xpm-dev  
php5 php5-gd php5-mysql php5-sqlite graphviz sqlite3
```

On systems such as CentOS and Fedora, the following packages may work:

```
# yum install php php-gd php-gettext php-mbstring php-mysql php-pdo php-  
sqlite php-xml graphviz graphviz-gd graphviz-php
```

It is difficult to anticipate the exact packages needed for all systems; however, searching your package manager for keywords (for example php sqlite) may help.

# How to do it...

We can install NagVis with an NDO backend as follows:

1. Download the latest sources for NagVis from <http://www.nagvis.org/downloads>:

```
$ wget http://www.nagvis.org/share/nagvis-1.8.5.tar.gz
```

2. Inflate .tar.gz and change into the uncompressed directory:

```
$ tar -xzf nagvis-1.8.5.tar.gz
$ cd nagvis-1.8.5
```

3. Run the install.sh script as root:

```
# ./install.sh
```

4. The script will attempt to find your Nagios Core installation and will ask you to specify a location for the new NagVis files. In our case, the defaults are correct and acceptable, so we can simply press *Enter*:

```
-- Checking paths --
Please enter the path to the nagios base directory [/usr/local/nagios]:
nagios path /usr/local/nagios found
Please enter the path to NagVis base [/usr/local/nagvis]:
```

5. The script will attempt to find all the prerequisites needed and will alert you if any are not found. If this is the case, you should abort the installation with *Ctrl + C* and install them before trying again:

```
-- Checking prerequisites --
PHP 5.6                                found
  PHP Module: gd 5.6                   found
  PHP Module: mbstring compiled_in    found
  PHP Module: gettext compiled_in     found
  PHP Module: session compiled_in     found
  PHP Module: xml compiled_in         found
  PHP Module: pdo compiled_in         found
  Apache mod_php                      found...
  PHP Module: sockets compiled_in
Graphviz 2.38
  Graphviz Module dot 2.38.0          found
  Graphviz Module neato 2.38.0        found
  Graphviz Module twopi 2.38.0        found
  Graphviz Module circo 2.38.0        found
  Graphviz Module fdp 2.38.0          found
SQLite 3.8                             found
```

6. The script will prompt you for an appropriate backend to configure as NagVis' data source. In this example, the only one we want is the ndo2db backend. Press n for all the others.

```
Checking Backends. (Available: mklivestatus,ndo2db,ido2db)
Do you want to use backend mklivestatus? [y]:
Do you want to use backend ndo2db? [n]:
Do you want to use backend ido2db? [n]:
```

**Livestatus Socket (/usr/local/nagios/var/rw/live)**

7. The script will attempt to detect your Apache HTTPD settings. It does a good job with most systems, but you should check the results are correct before you press Enter. It should also be safe to allow it to create an Apache configuration file for you.

**-- Trying to detect Apache settings --**

**Please enter the web path to NagVis [/nagvis]:**

**Please enter the name of the web-server user [www-data]:**

**Please enter the name of the web-server group [www-data]:**

**create Apache config file [y]:**

8. The script will give you a summary of its intentions to install the software and will ask you to confirm. Do so by pressing *Enter*:

**-- Summary --**

**NagVis home will be: /usr/local/nagvis**

**Owner of NagVis files will be: www-data**

**Group of NagVis files will be: www-data**

**Path to Apache config dir is: /etc/apache2/conf.d**

**Apache config will be created: yes**

**Installation mode: install**

**Do you really want to continue? [y]:**

9. Restart the Apache HTTPD server:

```
# apache2ctl configtest
```

```
# apache2ctl restart
```

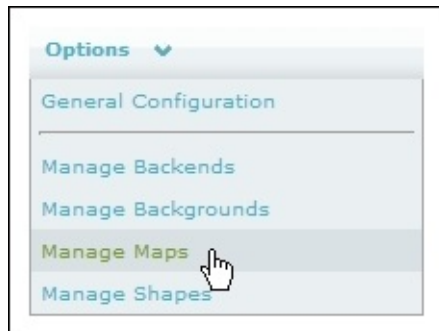
If all the preceding steps are taken correctly, once the installation is finished, you should be able to visit the NagVis configuration page on <http://olympus.example.net/nagvis/>, substituting the hostname for your own Nagios Core server.



You can then log in with the default username **admin** and the password **admin** to

take a look at some of the demo maps. There is a little more to be done before we can get our automap working:

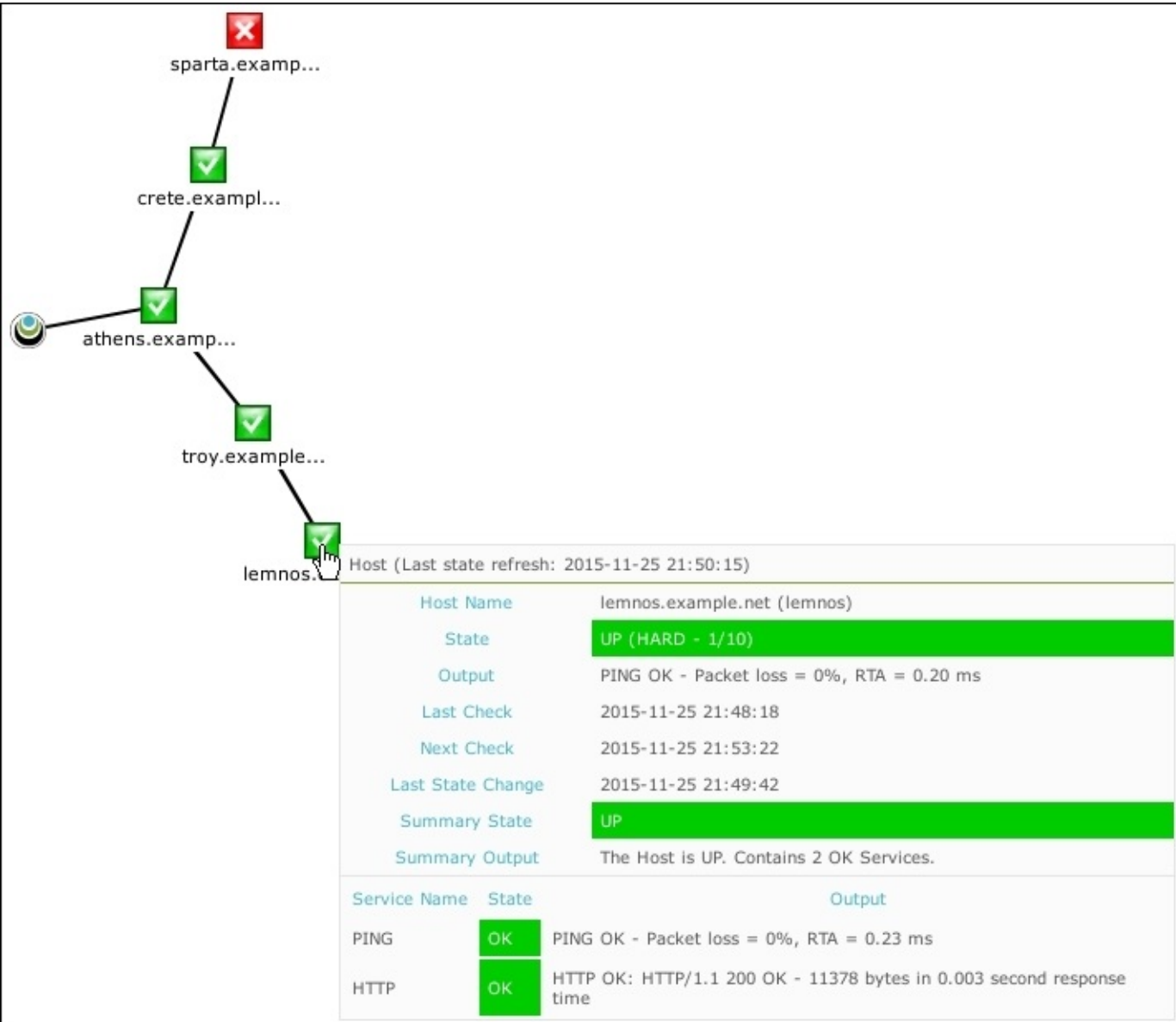
10. Log in to the NagVis web interface and click on **Manage Maps** under the **Options** menu.



11. Under **Create Map**, make the following settings:
  - For **ID (Internal Name)**, enter the **Automap** value.
  - For **Alias**, enter **Automap Test**. For **Map Type**, select **Automap based on parent/child relations**.
  - Then, click on **Create**.

A screenshot of the 'Create Map' form in the NagVis web interface. The form has three input fields: 'ID (Internal Name)' with the value 'Automap', 'Alias' with the value 'Automap Test', and 'Map Type' with a dropdown menu showing 'Automap based on paren'. Below the fields is a 'Create' button, which is being clicked by a mouse cursor.

With this done, the page should refresh and show you a map of your network, which is automatically generated from your configuration, in a similar style as the Nagios Core web interface status map. You should also be able to hover your mouse over individual nodes to see the details of the hosts.



## How it works...

NagVis' automap is generated from the data retrieved via the MK Livestatus backend. It generates the map in much the same way as the default status map in the Nagios Core web administration area does. However, it is much more versatile and more scalable for larger networks. The parent and child relationships defined in the configuration are used to make a tree-style map.

## There's more...

The uses of NagVis could fill an entire book and the automap is only one of many possible maps, including defining one's own backgrounds, icons, labels, and hover behavior. For more details on how to make customized maps as well as other styles of automap, consult the NagVis documentation at <http://www.nagvis.org/doc>.

## See also

- *Creating a network host hierarchy*, [Chapter 8](#), *Managing Network Layout*
- *Using the network map*, [Chapter 8](#), *Managing Network Layout*
- *The Reading a status from a Unix socket with MK Livestatus recipe* in this chapter
- *The Writing customized Nagios Core reports recipe* in this chapter





# Writing custom Nagios Core management scripts

As established in [Chapter 7, Using the Web Interface](#), some control over the Nagios Core process can be achieved by different areas of a web interface by enabling or disabling notifications or checks, scheduling downtime for hosts or services, or acknowledging existing programs. However, for large networks in particular, we may often need a more flexible approach to run operations on a large number of hosts or services in batches or to disable or enable features from another script, which is cumbersome to manage via the CGI.

For example, if we had a list of more than a hundred hosts in a segment of our network that were going to become inaccessible due to scheduled maintenance at a known time, it would take a very long time for us to set up all the scheduled downtime in the web interface to avoid sending out notifications for those hosts. It would be better to do this via a command line.

In this recipe, we'll install a custom script in `/usr/local/bin` on our monitoring server called `ncdt`, which is short for Nagios Core Downtime, that can be used to create downtime quickly for named hosts in the given time period. The downtime can be created using the `SCHEDULE_HOST_DOWNTIME` custom command that is written to the **external** commands file located at `/usr/local/nagios/var/rw/nagios.cmd`. We'll then use this script to loop over a list of hostnames to quickly set up an appropriate fixed scheduled downtime for all of them.

# Getting ready

You'll need to have a server running Nagios Core 4.0 or newer version. In this example, we'll provide the complete `ncdt` shell script as part of the instructions, but, to understand the script and the loop over the list of hosts, some experience with shell script would be helpful. On your monitoring server, you will also need GNU Bash 2.05a or newer, `date(1)` from GNU `sh-utils` 2.0 or newer, and the administrative rights to create new scripts in `/usr/local/bin`.

# How to do it...

We can create, test, and apply the `ncdt` shell script as follows:

1. Change the directory to `/usr/local/bin` on the monitoring server or some other directory in your path suitable for custom scripts like this one:

```
# cd /usr/local/bin
```

2. Edit a new script `ncdt`:

```
# vi ncdt
```

3. Paste the following shell script into the file:

```
#!/usr/bin/env bash

# Define some fields for the command
now=$(date +%s)
author=${SUDO_USER:-$USER}
fixed=1
trigger=0
duration=0
comment='Batch downtime set by ncdt'

# Read the hostname from the first argument
hostname=${1:?}

# Attempt to parse start and end dates; exit if the call doesn't work
dtstart=$(date +%s --date "${2:?}") || exit
dtend=$(date +%s --date "${3:?}") || exit

# Write command and print message if it fails; succeed silently
printf "[%lu] SCHEDULE_HOST_DOWNTIME;%s;%s;%s;%u;%u;%u;%s;%s\n" \
    "$now" "$hostname" "$dtstart" "$dtend" \
    "$fixed" "$trigger" "$duration" "$author" "$comment" \
    > /usr/local/nagios/var/rw/nagios.cmd
```

4. Make the script executable with `chmod(1)`:

```
# chmod +x ncdt
```

5. We can test the script by giving it three parameters in this order:

- A hostname for an existing host configured in Nagios Core
- A start time for the scheduled downtime
- An end time for the scheduled downtime

For this example, we'll set a downtime for the `sparta.example.net` host from “now” (the current time) until 5:00am tomorrow:

```
# ncdt sparta.example.net now '5:00am tomorrow'
```

6. We need to check whether the downtime actually works, which we can do by looking at the contents of the log file; in this case, `/usr/local/nagios/var/nagios.log`:

```
[1448444263] EXTERNAL COMMAND:  
SCHEDULE_HOST_DOWNTIME;sparta.example.net; 1448444263;  
1448467200;1;0;0;tom;Batch downtime set by ncdt  
[1448444263] HOST DOWNTIME ALERT: sparta.example.net;STARTED; Host has  
entered a period of scheduled downtime
```

In this example, the output shows that the external command went through correctly and that the host immediately entered downtime, as expected.

7. We can now loop over a list of hostnames in the `downtime-list` file to quickly set the same period of downtime for a list of any number of hosts, using a `while-read` loop:

```
# while read -r hostname ; do  
> ncdt "$hostname" now '5:00am tomorrow'  
> done < downtime-list
```

## How it works...

The Bash script accepts three arguments: the hostname of the host that should enter downtime, the date and time the downtime should start, and the date and time it should end. The dates can be provided in a variety of formats and keywords understood by the GNU `date(1)`; they are translated to Unix timestamps.

The script fails if any of these are not set. It calculates the current timestamp for the command time, then the name of the user writing the command, and uses fixed values for the remaining fields of the `SCHEDULE_HOST_DOWNTIME` command. It formats the complete command with all its arguments with `printf`, separating them with semicolons, and it writes the completed command to the external commands file to be processed by Nagios Core.

Once we know that the script is working, we're able to apply it in the `for` or `while` loops to quickly run the same script over a large number of hosts, perhaps in a file or a list pasted into the terminal window.

## There's more...

Custom scripts don't need to be written in shell script. They can be written in any programming language that suits, as the only filesystem interaction needed is to write a line to the external commands file. We could, for example, write scripts similar to `ncdt` in Perl, Python, or Ruby.

Your scripts could have more features than this or be more user friendly; some ideas for expanding on this simple script might include the following:

- Manual pages for `man(1)` or help output when run with `--help`
- Working on services instead of hosts
- Checking whether the host exists with MK Livestatus before running the command
- Verifying that the command worked, for example, the downtime or acknowledgement was established correctly in Nagios Core

Here are some other examples of tasks that might benefit from script shortcuts. Note that some of these would require NDOUtils or MK Livestatus as a backend to retrieve data from Nagios Core.

- Acknowledging problems with hosts or services
- Searching for Nagios hostnames matching a pattern
- Viewing the most recent notifications for a given host
- Sending custom notifications to all the contacts for a given host
- Enabling or disabling notifications
- Enabling or disabling active checks
- Enabling or disabling event handlers

### Tip

A complete list of the commands available for the external commands file is available at <https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/extcommands.html>.

## See also

- The *Reading a status from a Unix socket with MK Livestatus* recipe in this chapter
- The *Reading a status into a database with NDOUtils* recipe in this chapter
- The *Writing customized Nagios Core reports* recipe in this chapter



# Index

## A

- abstraction
  - about / [Introduction](#)
- active checks
  - about / [Introduction](#)
- allowed client hosts
  - setting, for NRPE / [Setting allowed client hosts for NRPE](#), [Getting ready](#), [How to do it...](#)
- alternative check command
  - using, for hosts / [Using an alternative check command for hosts](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- alternative SSH port
  - checking / [Checking an alternative SSH port](#), [How to do it...](#), [There's more...](#)
- Apache
  - URL / [Getting ready](#)
- authenticated contacts
  - using / [Using authenticated contacts](#), [How to do it...](#), [See also](#)
- authentication
  - used, for Nagios Core web interface / [Using authentication for the Nagios Core web interface](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- automap
  - about / [Getting extra visualizations with NagVis](#)
- availability reports
  - viewing / [Viewing and interpreting availability reports](#), [Getting started](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - interpreting / [Viewing and interpreting availability reports](#), [Getting started](#), [How to do it...](#), [How it works...](#), [There's more...](#)

# B

- brief outages
  - managing, with flapping / [Managing brief outages with flapping](#), [How to do it...](#), [How it works...](#)

# C

- Cacti
  - URL / [There's more...](#)
- check\_by\_ssh
  - about / [Introduction](#)
  - used, with key authentication / [Using check\\_by\\_ssh with key authentication instead of NRPE](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- check\_http command
  - about / [Monitoring web services](#)
- check\_http plugin
  - using / [There's more...](#), [Checking that a website returns a given string](#), [There's more...](#)
- check\_mk
  - about / [Introduction](#)
  - using / [Using check\\_mk instead of NRPE](#), [How to do it...](#), [How it works...](#)
  - URL / [Using check\\_mk instead of NRPE](#), [How to do it...](#), [There's more...](#)
- check\_mysql plugin
  - about / [Monitoring database services](#)
- check\_nrpe
  - about / [Introduction](#)
- check\_period property
  - about / [How it works...](#)
- check\_ping plugin
  - about / [Monitoring PING for any host](#)
- check\_rsync
  - URL / [Getting ready](#)
- check\_snmp
  - about / [Introduction](#)
- check\_snmp plugin
  - using / [Monitoring the output of an SNMP query](#)
- check\_ssh plugin
  - about / [Monitoring SSH for any host](#)
- child host
  - about / [How it works...](#)
- cluster
  - nodes, monitoring / [Monitoring individual nodes in a cluster](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- command
  - about / [Introduction](#)
  - creating / [Creating a new command](#), [How to do it...](#), [How it works...](#)
- command arguments
  - about / [Changing thresholds for PING RTT and packet loss](#)

- command definitions
  - creating, for NRPE / [Creating new NRPE command definitions securely](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- comments
  - adding, on services / [Getting started](#), [How to do it...](#), [There's more...](#)
  - adding, on hosts / [Getting started](#), [How to do it...](#), [There's more...](#)
- Comprehensive Perl Archive Network (CPAN)
  - URL / [Getting ready](#)
  - using / [Getting ready](#)
- configuration
  - viewing, in Nagios Core web interface / [Viewing configuration in the web interface](#), [How to do it...](#), [How it works...](#)
  - simplifying, inheritance used / [Using inheritance to simplify configuration](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- configuration, Nagios Core
  - hosts / [Introduction](#)
  - services / [Introduction](#)
  - contacts / [Introduction](#)
  - verifying / [Verifying configuration](#), [How to do it...](#), [How it works...](#)
- configuration directory
  - placing, under version control / [Keeping a configuration under version control](#), [Getting ready](#), [How it works...](#), [There's more...](#)
- configuration files
  - grouping, in directories / [Grouping configuration files in directories](#), [How to do it...](#), [There's more...](#)
- contactgroup
  - creating / [Creating a new contactgroup](#), [How to do it...](#), [There's more...](#)
- contact rotation
  - automating / [Automating contact rotation](#), [How to do it...](#), [How it works...](#)
- contacts
  - about / [Introduction](#)
- custom directives
  - using / [Using custom directives](#), [How to do it...](#), [There's more...](#)
  - reference link / [There's more...](#)
- customized Nagios Core reports
  - writing / [Writing customized Nagios Core reports](#), [How to do it...](#), [How it works...](#)
- custom Nagios Core management scripts
  - writing / [Writing custom Nagios Core management scripts](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- custom notification method
  - defining / [Defining a custom notification method](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- custom object variables

- about / [How it works...](#)

# D

- database services
  - monitoring / [Monitoring database services](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- debugging information
  - writing, to log file / [Writing debugging information to the Nagios log file](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- directories
  - configuration files, grouping / [Grouping configuration files in directories](#), [How to do it...](#), [There's more...](#)
- disk usage
  - thresholds, modifying / [Changing thresholds for disk usage](#), [How it works...](#), [There's more...](#)
- distributed monitoring setup
  - about / [There's more...](#)

# E

- e-mail contact
  - creating / [Creating a new e-mail contact](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- environment variables
  - about / [Using macros as environment variables in a plugin](#)
- escalation
  - defining, for repeated notifications / [Defining an escalation for repeated notifications](#), [How to do it...](#), [There's more...](#), [See also](#)
- event handlers
  - about / [Introduction](#)
- event handler script
  - setting up / [Setting up an event handler script](#), [How it works...](#), [There's more...](#)
- existing command
  - customizing / [Customizing an existing command](#), [How to do it...](#), [There's more...](#)
- external command file
  - about / [Introduction](#)
- external commands file
  - about / [Writing custom Nagios Core management scripts](#)
  - URL / [There's more...](#)

# F

- fixed downtime
  - defining / [There's more...](#)
- flapping
  - brief outages, managing / [Managing brief outages with flapping](#), [How to do it...](#), [How it works...](#)
  - reference link / [How it works...](#)
- flapping percentage thresholds
  - adjusting, for service / [Getting ready](#), [How it works...](#), [There's more...](#)
  - adjusting, for host / [Getting ready](#), [How it works...](#), [There's more...](#)



# G

- GNU m4(1)
  - URL / [Getting ready](#)
- groups
  - about / [Introduction](#)
  - host roles, configuring / [Configuring host roles using groups](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - building, regular expressions used / [Building groups using regular expressions](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- guest virtual machine
  - about / [Getting ready](#)

# H

- hardware device
  - monitoring / [Monitoring a RAID or other hardware device](#), [How it works...](#)
- host
  - check frequency, customizing / [Specifying how frequently to check a host or service](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - downtime, scheduling / [Scheduling downtime for a host or service](#), [How to do it...](#), [How it works...](#)
  - flapping percentage thresholds, adjusting / [Getting ready](#), [How it works...](#), [There's more...](#)
  - PING, monitoring / [Monitoring PING for any host](#), [How to do it...](#), [There's more...](#)
  - SSH, monitoring / [Monitoring SSH for any host](#), [Getting ready](#), [How it works...](#), [There's more...](#)
  - states, tracking with Nagiosgraph / [Tracking host and service states with Nagiosgraph](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
- host definitions
  - building, dynamically / [Dynamically building host definitions](#), [How to do it...](#), [There's more...](#)
- host dependency
  - establishing / [Establishing a host dependency](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- host escalation
  - about / [Defining an escalation for repeated notifications](#)
- hostgroup
  - creating / [Creating a new hostgroup](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - services, executing / [Running a service on all hosts on a group](#), [Getting ready](#), [How it works...](#)
  - about / [There's more...](#), [There's more...](#)
- host physical machine
  - about / [Getting ready](#)
- host reachability logic
  - about / [Introduction](#)
- host roles
  - configuring, with groups / [Configuring host roles using groups](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- hosts
  - about / [Introduction](#)
  - alternative check command, using / [Using an alternative check command for hosts](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
  - icons, selecting / [Choosing icons for hosts](#), [How to do it...](#), [There's more...](#)
- host template

- about / [There's more...](#)
- HTTP service
  - creating / [Creating a new HTTP service](#), [How to do it...](#), [How it works...](#), [There's more...](#)

# I

- icons
  - selecting, for hosts / [Choosing icons for hosts](#), [How to do it...](#), [There's more...](#)
- inheritance
  - used, for simplifying configuration / [Using inheritance to simplify configuration](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - reference link / [There's more...](#)

# K

- key authentication
  - check\_by\_ssh, using / [Using check\\_by\\_ssh with key authentication instead of NRPE](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)

# L

- libgd-tools package
  - URL / [Getting ready](#)
- Liferea
  - URL / [How to do it...](#)
- limited sudo(8) privileges
  - providing, to NRPE / [Giving limited sudo\(8\) privileges to NRPE](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- listening address
  - setting, for NRPE / [Setting the listening address for NRPE](#), [Getting ready](#), [How to do it...](#)
- Livestatus Query Language (LQL)
  - about / [How it works...](#)
- local services
  - monitoring, on remote machine with NRPE / [Monitoring local services on a remote machine with NRPE](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- log file
  - debugging information, writing to / [Writing debugging information to the Nagios log file](#), [How to do it...](#), [How it works...](#), [There's more...](#)

# M

- m4
  - URL, for Nagios configuration / [There's more...](#)
- macro environment variables
  - about / [How it works...](#)
- macro expansion
  - about / [How to do it...](#)
- macros
  - used, as environment variables in plugin / [Using macros as environment variables in a plugin](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - about / [Introduction](#)
  - \$USER1\$ / [How it works...](#)
  - \$HOSTADDRESS\$ / [How it works...](#)
  - \$ARG1\$ / [How it works...](#)
  - defining, in resource file / [Defining macros in a resource file](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- mail services
  - monitoring / [Monitoring mail services](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- management interface
  - about / [Using custom directives](#)
- MK Livestatus
  - about / [Introduction](#)
  - used, for reading status from Unix socket / [Reading status from a Unix socket with MK Livestatus](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - URL / [How to do it...](#), [There's more...](#)
- Multi-Router Traffic Grapher (mrtg)
  - about / [Monitoring Nagios performance with nagiosstats](#)
  - URL / [Getting ready](#)
- MySQL database
  - status, reading with NDOUtils / [Reading status in a MySQL database with NDOUtils](#), [Getting ready](#), [How to do it...](#), [How it works...](#)

# N

- Nagios Core
  - about / [Introduction](#)
  - URL / [Getting ready](#), [Getting ready](#)
  - URL, for quick start guide / [Getting ready](#)
  - URL, for documentation / [Introduction](#)
- Nagios Core 4.0
  - URL / [How it works...](#)
- Nagios Core Plugins
  - URL / [There's more...](#)
  - reference link / [There's more...](#)
- Nagios Core server
  - guidelines, for securing / [Introduction](#)
- Nagios Core web interface
  - about / [Introduction](#)
  - comments, adding on hosts / [Getting started](#), [How to do it...](#), [There's more...](#)
  - comments, adding on services / [Getting started](#), [How to do it...](#), [There's more...](#)
  - configuration, viewing / [Viewing configuration in the web interface](#), [How to do it...](#), [How it works...](#)
  - checks, scheduling / [Scheduling checks from the web interface](#), [Getting started](#), [How to do it...](#), [How it works...](#), [There's more](#)
  - problem, acknowledging / [Acknowledging a problem via the web interface](#), [How to do it...](#), [There's more...](#)
  - authentication, used for / [Using authentication for the Nagios Core web interface](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- Nagios Exchange
  - URL / [There's more...](#), [There's more...](#), [Introduction](#), [How to do it...](#), [There's more...](#)
- Nagiosgraph
  - about / [Introduction](#)
  - used, for tracking host and service states / [Tracking host and service states with Nagiosgraph](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
  - URL / [Getting ready](#)
- Nagios Plugins
  - about / [Introduction](#)
  - URL / [Introduction](#), [Introduction](#)
  - URL, for Development Guidelines / [There's more...](#)
- Nagios Plugins package
  - URL / [How to do it...](#)
- Nagios Service Check Acceptor (NSCA)
  - about / [Introduction](#)
- nagiosstats utility
  - used, for monitoring Nagios performance / [Monitoring Nagios performance with](#)



[nagiosstats](#), [How to do it...](#), [How it works...](#), [There's more...](#)

- NagVis
  - about / [Introduction](#)
  - used, for visualizations / [Getting extra visualizations with NagVis](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - URL / [How to do it...](#), [There's more...](#)
  - reference link / [How to do it...](#)
- NagVis extension
  - about / [There's more...](#)
- named pipe
  - about / [How it works...](#)
- NDOUtils
  - about / [Introduction](#)
  - used, for reading status in MySQL database / [Reading status in a MySQL database with NDOUtils](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
  - URL / [How to do it...](#), [There's more...](#)
- Net-SNMP
  - URL / [There's more...](#), [Getting ready](#)
- network host
  - creating / [Creating a new network host](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- network host hierarchy
  - creating / [Creating a network host hierarchy](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- network map
  - about / [Introduction](#)
  - using / [Using the network map](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - using, as overlay / [Using the network map as an overlay](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- network structure, approaches
  - host parent definitions / [Introduction](#)
  - host and service dependencies / [Introduction](#)
- nodes
  - monitoring, in cluster / [Monitoring individual nodes in a cluster](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- notification history
  - viewing / [Viewing and interpreting notification history](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - interpreting / [Viewing and interpreting notification history](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- notifications
  - about / [Introduction](#)
  - time periods, configuring / [Configuring notification periods](#), [How to do it...](#),

### [How it works...](#)

- configuring, for groups / [Configuring notifications for groups](#), [How to do it...](#), [How it works...](#)
- states, selecting / [Choosing states for notification](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- number of failed checks, specifying / [Specifying the number of failed checks before notification](#), [How to do it...](#), [How it works...](#), [See also](#)
- filtering, based on host importance / [Filtering notifications based on a host or service value](#), [How to do it...](#), [There's more...](#)
- filtering, based on service importance / [Filtering notifications based on a host or service value](#), [How to do it...](#), [There's more...](#)
- notification\_period property
  - about / [How it works...](#)
- NRPE
  - about / [Introduction](#)
  - local services, monitoring on remote machine / [Monitoring local services on a remote machine with NRPE](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - URL / [How to do it...](#)
  - listening address, setting / [Setting the listening address for NRPE](#), [Getting ready](#), [How to do it...](#)
  - allowed client hosts, setting / [Setting allowed client hosts for NRPE](#), [Getting ready](#), [How to do it...](#)
  - command definitions, creating / [Creating new NRPE command definitions securely](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - limited sudo(8) privileges, providing / [Giving limited sudo\(8\) privileges to NRPE](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- NSCA
  - passive checks, submitting from remote host / [Submitting passive checks from a remote host with NSCA](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - URL / [How to do it...](#)

# O

- object definitions
  - reference link / [There's more...](#)
- objects directives
  - another objects directives, used in check / [Using another object's directives in a host or service check](#), [How to do it...](#), [There's more...](#)
- on-demand macros
  - about / [Using another object's directives in a host or service check](#)
  - reference link / [There's more...](#)

# P

- packet loss
  - thresholds, modifying / [Changing thresholds for PING RTT and packet loss](#), [How to do it...](#), [There's more...](#)
- parent host
  - about / [There's more...](#), [Getting ready](#), [How it works...](#)
- passive checks
  - about / [Introduction](#)
  - allowing / [Allowing and submitting passive checks](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - submitting / [Allowing and submitting passive checks](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - submitting, from remote host with NSCA / [Submitting passive checks from a remote host with NSCA](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - submitting, in response to SNMP traps / [Submitting passive checks in response to SNMP traps](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- Performance Data feature
  - reference link / [There's more...](#)
- PING
  - monitoring, for any host / [Monitoring PING for any host](#), [How to do it...](#), [There's more...](#)
- PING RTT
  - thresholds, modifying / [Changing thresholds for PING RTT and packet loss](#), [How to do it...](#), [There's more...](#)
- plugin
  - about / [Introduction](#)
  - searching / [Finding a plugin](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - installing / [Installing a plugin](#), [How to do it...](#), [How it works...](#)
  - removing / [Getting ready](#), [How it works...](#)
  - writing, from scratch / [Writing a new plugin from scratch](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - threshold checks, implementing / [Implementing threshold checks in a plugin](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - macros, used as environment variables / [Using macros as environment variables in a plugin](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- PNP4Nagios
  - URL / [There's more...](#)
- private key
  - about / [How to do it...](#)
- Pro Git
  - reference link / [How it works...](#)
- public key

- about / [How to do it...](#)
- public key cryptography
  - reference link / [Getting ready](#)

# R

- RAID
  - monitoring / [Monitoring a RAID or other hardware device](#), [How it works...](#)
- reachability logic
  - about / [How it works...](#)
- redundant monitoring host
  - setting up / [Setting up a redundant monitoring host](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
  - reference link / [There's more...](#)
- regular expressions
  - used, for building groups / [Building groups using regular expressions](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
  - URL / [Getting ready](#)
- remote host
  - passive checks, submitting with NSCA / [Submitting passive checks from a remote host with NSCA](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- remote machine
  - local services, monitoring with NRPE / [Monitoring local services on a remote machine with NRPE](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- repeated notifications
  - escalation, defining / [Defining an escalation for repeated notifications](#), [How to do it...](#), [There's more...](#), [See also](#)
- resource file
  - macros, defining / [Defining macros in a resource file](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- return values
  - about / [There's more...](#)
- round trip time (RTT)
  - about / [How to do it...](#)
- rsync
  - URL / [There's more...](#)

# S

- service
  - check frequency, customizing / [Specifying how frequently to check a host or service](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - downtime, scheduling / [Scheduling downtime for a host or service](#), [How to do it...](#), [How it works...](#)
  - flapping percentage thresholds, adjusting / [Getting ready](#), [How it works...](#), [See also](#)
  - states, tracking with Nagiosgraph / [Tracking host and service states with Nagiosgraph](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
- Service and Host Freshness Checks
  - reference link / [There's more...](#)
- service dependency
  - establishing / [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- service escalation
  - about / [Defining an escalation for repeated notifications](#)
- servicegroup
  - creating / [Creating a new servicegroup](#), [How to do it...](#), [How it works...](#)
- services
  - about / [Introduction](#)
  - executing, on hostgroup / [Running a service on all hosts on a group](#), [Getting ready](#), [How it works...](#)
- Simple Network Management Protocol (SNMP)
  - about / [Introduction](#), [Getting ready](#)
- SNMP OID
  - creating / [Creating an SNMP OID for monitoring](#), [How to do it...](#), [There's more...](#)
- SNMP query
  - output, monitoring / [Monitoring the output of an SNMP query](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- SNMP traps
  - passive checks, submitting / [Submitting passive checks in response to SNMP traps](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- SNMPTT
  - URL / [Getting ready](#)
- SSH
  - monitoring, for any host / [Monitoring SSH for any host](#), [Getting ready](#), [How it works...](#), [There's more...](#)
- SSH authentication
  - reference link / [Getting ready](#)
- submit\_check\_result event handler
  - reference link / [Getting ready](#)

# T

- Tactical Overview
  - using / [Using the Tactical Overview](#), [How to do it...](#), [How it works...](#)
- template
  - about / [There's more...](#)
- templates
  - about / [Introduction](#)
- threshold checks
  - implementing, in plugin / [Implementing threshold checks in a plugin](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- threshold format syntax
  - URL / [How it works...](#)
- thresholds
  - modifying, for PING RTT / [Changing thresholds for PING RTT and packet loss](#), [How to do it...](#), [There's more...](#)
  - modifying, for packet loss / [Changing thresholds for PING RTT and packet loss](#), [How to do it...](#), [There's more...](#)
  - modifying, for disk usage / [Changing thresholds for disk usage](#), [How it works...](#), [There's more...](#)
- time period
  - creating / [Creating a new time period](#), [How to do it...](#), [There's more...](#)
- trends
  - interpreting / [Viewing and interpreting trends](#), [Getting started](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - viewing / [Viewing and interpreting trends](#), [Getting started](#), [How to do it...](#), [How it works...](#), [There's more...](#)



# U

- Unix socket
  - status, reading with MK Livestatus / [Reading status from a Unix socket with MK Livestatus](#), [How to do it...](#), [How it works...](#), [There's more...](#)

# V

- version control
  - configuration directory, placing / [Keeping a configuration under version control](#), [Getting ready](#), [How it works...](#), [There's more...](#)
- visualizations
  - with NagVis / [Getting extra visualizations with NagVis](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)

# W

- weather map
  - about / [Introduction](#)
- web services
  - monitoring / [Monitoring web services](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- website
  - checking, for string response / [Checking that a website returns a given string](#), [How to do it...](#), [How it works...](#)
- wrapper script
  - about / [How it works...](#)