Centreon

Poller States	Hosts	Up	Down	Unreachable	Pending	Services	Ok	Warning	Critical	Unknown	Pending
	208	207	1	0	0	3652	3368	12/23	56/179	72/82	0

Documentation - You are service.desk Logout

Monitoring Views Reporting

Services | Hosts | Event Logs

- By Status
  - Unhandled Problems
  - Service Problems
  - All Services
- By Host
  - Details
  - Summary
- By Host Group
  - Details
  - Summary
- By Service Group
  - Details
    - Problems
    - Acknowledged
    - Not Acknowledged
  - Summary
- Meta Services
  - Meta Services
- Nagios
  - Scheduling Queue
  - Downtime
  - Comments

Monitoring ▸ Services ▸ Details 2013/05/11 22:27

Nagios®
Main Configuration File Options

Up To: Contents

Notes

When creating and/or editing configuration files, keep the following in mind:

1. Lines that start with a '#' character are taken to be comments and are not processed
2. Variables names must begin at the start of the line - no white space is allowed before the name
3. Variable names are case-sensitive

Sample Configuration File

Tip: A sample main configuration file (*/usr/local/nagios/etc/nagios.cfg*) is installed for you when you follow the quickstart installation guide.

Config File Location

The main configuration file is usually named *nagios.cfg* and located in the */usr/local/nagios/etc/* directory.

Configuration File Variables

Below you will find descriptions of each main Nagios configuration file option...

Log File

Format:	**log_file=<file_name>**
Example:	**log_file=/usr/local/nagios/var/nagios.log**

This variable specifies where Nagios should create its main log file. This should be the first variable that you define in your configuration file, as Nagios will try to write errors that it finds in the rest of your configuration data to this file. If you have log rotation enabled, this file will automatically be rotated every hour, day, week, or month.

Object Configuration File

Format:	**cfg_file=<file_name>**
Example:	**cfg_file=/usr/local/nagios/etc/hosts.cfg** **cfg_file=/usr/local/nagios/etc/services.cfg** **cfg_file=/usr/local/nagios/etc/commands.cfg**

This directive is used to specify an object configuration file containing object definitions that Nagios should use for monitoring. Object configuration files contain definitions for hosts, host groups, contacts, contact groups, services, commands, etc. You can seperate your configuration information into several files and specify multiple *cfg_file=* statements to have each of them processed.

Object Configuration Directory

Format:	**cfg_dir=<directory_name>**
Example:	**cfg_dir=/usr/local/nagios/etc/commands** **cfg_dir=/usr/local/nagios/etc/services** **cfg_dir=/usr/local/nagios/etc/hosts**

This directive is used to specify a directory which contains object configuration files that Nagios should use for monitoring. All files in the directory with a *.cfg* extension are processed as object config files. Additionally, Nagios will recursively process all config files in subdirectories of the directory you specify here. You can seperate your configuration files into different directories and specify multiple *cfg_dir=* statements to have all config files in each directory processed.

Object Cache File

Format:	**object_cache_file=<file_name>**
Example:	**object_cache_file=/usr/local/nagios/var/objects.cache**

This directive is used to specify a file in which a cached copy of object definitions should be stored. The cache file is (re)created every time Nagios is (re)started and is used by the CGIs. It is intended to speed up config file caching in the CGIs and allow you to edit the source object config files while Nagios is running without affecting the output displayed in the CGIs.

Precached Object File

Format:	**precached_object_file=<file_name>**
Example:	**precached_object_file=/usr/local/nagios/var/objects.precache**

This directive is used to specify a file in which a pre-processed, pre-cached copy of object definitions should be stored. This file can be used to drastically improve startup times in large/complex Nagios installations. Read more information on how to speed up start times here.

Resource File

Format:	**resource_file=<file_name>**
Example:	**resource_file=/usr/local/nagios/etc/resource.cfg**

This is used to specify an optional resource file that can contain $USERn$ macro definitions. $USERn$ macros are useful for storing usernames, passwords, and items commonly used in command definitions (like directory paths). The CGIs will *not* attempt to read resource files, so you can set restrictive permissions (600 or 660) on them to protect sensitive information. You can include multiple resource files by adding multiple resource_file statements to the main config file - Nagios will process them all. See the sample resource.cfg file in the *sample-config/* subdirectory of the Nagios distribution for an example of how to define $USERn$ macros.

Temp File

Format:	**temp_file=<file_name>**
Example:	**temp_file=/usr/local/nagios/var/nagios.tmp**

This is a temporary file that Nagios periodically creates to use when updating comment data, status data, etc. The file is deleted when it is no longer needed.

Temp Path

Format:	**temp_path=<dir_name>**
Example:	**temp_path=/tmp**

This is a directory that Nagios can use as scratch space for creating temporary files used during the monitoring process. You should run *tmpwatch*, or a similiar utility, on this directory occassionally to delete files older than 24 hours.

Status File

Format:	**status_file=<file_name>**
Example:	**status_file=/usr/local/nagios/var/status.dat**

This is the file that Nagios uses to store the current status, comment, and downtime information. This file is used by the CGIs so that current monitoring status can be reported via a

web interface. The CGIs must have read access to this file in order to function properly. This file is deleted every time Nagios stops and recreated when it starts.

Status File Update Interval

Format:	**status_update_interval=<seconds>**
Example:	**status_update_interval=15**

This setting determines how often (in seconds) that Nagios will update status data in the status file. The minimum update interval is 1 second.

Nagios User

Format:	**nagios_user=<username/UID>**
Example:	**nagios_user=nagios**

This is used to set the effective user that the Nagios process should run as. After initial program startup and before starting to monitor anything, Nagios will drop its effective privileges and run as this user. You may specify either a username or a UID.

Nagios Group

Format:	**nagios_group=<groupname/GID>**
Example:	**nagios_group=nagios**

This is used to set the effective group that the Nagios process should run as. After initial program startup and before starting to monitor anything, Nagios will drop its effective privileges and run as this group. You may specify either a groupname or a GID.

Notifications Option

Format:	**enable_notifications=<0/1>**
Example:	**enable_notifications=1**

This option determines whether or not Nagios will send out notifications when it initially (re)starts. If this option is disabled, Nagios will not send out notifications for any host or service. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Disable notifications
- 1 = Enable notifications (default)

Service Check Execution Option

Format:	**execute_service_checks=<0/1>**
Example:	**execute_service_checks=1**

This option determines whether or not Nagios will execute service checks when it initially (re)starts. If this option is disabled, Nagios will not actively execute any service checks and will remain in a sort of "sleep" mode (it can still accept passive checks unless you've disabled them). This option is most often used when configuring backup monitoring servers, as described in the documentation on redundancy, or when setting up a distributed monitoring environment. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Don't execute service checks
- 1 = Execute service checks (default)

Passive Service Check Acceptance Option

Format:	**accept_passive_service_checks=<0/1>**
Example:	**accept_passive_service_checks=1**

This option determines whether or not Nagios will accept passive service checks when it initially (re)starts. If this option is disabled, Nagios will not accept any passive service checks. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Don't accept passive service checks
- 1 = Accept passive service checks (default)

Host Check Execution Option

Format:	**execute_host_checks=<0/1>**
Example:	**execute_host_checks=1**

This option determines whether or not Nagios will execute on-demand and regularly scheduled host checks when it initially (re)starts. If this option is disabled, Nagios will not actively execute any host checks, although it can still accept passive host checks unless you've disabled them). This option is most often used when configuring backup monitoring servers, as described in the documentation on redundancy, or when setting up a distributed monitoring environment. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Don't execute host checks
- 1 = Execute host checks (default)

Passive Host Check Acceptance Option

Format:	**accept_passive_host_checks=<0/1>**
Example:	**accept_passive_host_checks=1**

This option determines whether or not Nagios will accept passive host checks when it initially (re)starts. If this option is disabled, Nagios will not accept any passive host checks. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Don't accept passive host checks
- 1 = Accept passive host checks (default)

Event Handler Option

Format:	**enable_event_handlers=<0/1>**
Example:	**enable_event_handlers=1**

This option determines whether or not Nagios will run event handlers when it initially (re)starts. If this option is disabled, Nagios will not run any host or service event handlers. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface. Values are as follows:

- 0 = Disable event handlers
- 1 = Enable event handlers (default)

Log Rotation Method

Format:	**log_rotation_method=<n/h/d/w/m>**
Example:	**log_rotation_method=d**

This is the rotation method that you would like Nagios to use for your log file. Values are as follows:

- n = None (don't rotate the log - this is the default)
- h = Hourly (rotate the log at the top of each hour)
- d = Daily (rotate the log at midnight each day)
- w = Weekly (rotate the log at midnight on Saturday)
- m = Monthly (rotate the log at midnight on the last day of the month)

Log Archive Path

Format:	**log_archive_path=<path>**
Example:	**log_archive_path=/usr/local/nagios/var/archives/**

This is the directory where Nagios should place log files that have been rotated. This option is ignored if you choose to not use the log rotation functionality.

External Command Check Option

Format:	**check_external_commands=<0/1>**
Example:	**check_external_commands=1**

This option determines whether or not Nagios will check the command file for commands that should be executed. This option must be enabled if you plan on using the command CGI to issue commands via the web interface. More information on external commands can be found here.

- 0 = Don't check external commands
- 1 = Check external commands (default)

External Command Check Interval

Format:	**command_check_interval=<xxx>[s]**
Example:	**command_check_interval=1**

If you specify a number with an "s" appended to it (i.e. 30s), this is the number of *seconds* to wait between external command checks. If you leave off the "s", this is the number of "time units" to wait between external command checks. Unless you've changed the interval_length value (as defined below) from the default value of 60, this number will mean minutes.

Note: By setting this value to **-1**, Nagios will check for external commands as often as possible. Each time Nagios checks for external commands it will read and process all commands present in the command file before continuing on with its other duties. More information on external commands can be found here.

External Command File

Format:	**command_file=<file_name>**
Example:	**command_file=/usr/local/nagios/var/rw/nagios.cmd**

This is the file that Nagios will check for external commands to process. The command CGI writes commands to this file. The external command file is implemented as a named pipe (FIFO), which is created when Nagios starts and removed when it shuts down. If the file exists when Nagios starts, the Nagios process will terminate with an error message. More information on external commands can be found here.

External Command Buffer Slots

Format:	**external_command_buffer_slots=<#>**
Example:	**external_command_buffer_slots=512**

Note: This is an advanced feature. This option determines how many buffer slots Nagios will reserve for caching external commands that have been read from the external command file by a worker thread, but have not yet been processed by the main thread of the Nagios deamon. Each slot can hold one external command, so this option essentially determines how many commands can be buffered. For installations where you process a large number of passive checks (e.g. distributed setups), you may need to increase this number. You should consider using MRTG to graph Nagios' usage of external command buffers. You can read more on how to configure graphing here.

Lock File

Format:	**lock_file=<file_name>**
Example:	**lock_file=/tmp/nagios.lock**

This option specifies the location of the lock file that Nagios should create when it runs as a daemon (when started with the -d command line argument). This file contains the process id (PID) number of the running Nagios process.

State Retention Option

Format:	**retain_state_information=<0/1>**
Example:	**retain_state_information=1**

This option determines whether or not Nagios will retain state information for hosts and services between program restarts. If you enable this option, you should supply a value for the state_retention_file variable. When enabled, Nagios will save all state information for hosts and service before it shuts down (or restarts) and will read in previously saved state information when it starts up again.

- 0 = Don't retain state information
- 1 = Retain state information (default)

State Retention File

Format:	**state_retention_file=<file_name>**
Example:	**state_retention_file=/usr/local/nagios/var/retention.dat**

This is the file that Nagios will use for storing status, downtime, and comment information before it shuts down. When Nagios is restarted it will use the information stored in this file for setting the initial states of services and hosts before it starts monitoring anything. In order to make Nagios retain state information between program restarts, you must enable the retain_state_information option.

Automatic State Retention Update Interval

Format:	**retention_update_interval=<minutes>**
Example:	**retention_update_interval=60**

This setting determines how often (in minutes) that Nagios will automatically save retention data during normal operation. If you set this value to 0, Nagios will not save retention data at regular intervals, but it will still save retention data before shutting down or restarting. If you have disabled state retention (with the retain_state_information option), this option has no effect.

Use Retained Program State Option

Format:	**use_retained_program_state=<0/1>**
Example:	**use_retained_program_state=1**

This setting determines whether or not Nagios will set various program-wide state variables based on the values saved in the retention file. Some of these program-wide state variables that are normally saved across program restarts if state retention is enabled include the enable_notifications, enable_flap_detection, enable_event_handlers, execute_service_checks, and accept_passive_service_checks options. If you do not have state retention enabled, this option has no effect.

- 0 = Don't use retained program state
- 1 = Use retained program state (default)

Use Retained Scheduling Info Option

Format:	**use_retained_scheduling_info=<0/1>**
Example:	**use_retained_scheduling_info=1**

This setting determines whether or not Nagios will retain scheduling info (next check times) for hosts and services when it restarts. If you are adding a large number (or percentage) of hosts and services, I would recommend disabling this option when you first restart Nagios, as it can adversely skew the spread of initial checks. Otherwise you will probably want to leave it enabled.

- 0 = Don't use retained scheduling info
- 1 = Use retained scheduling info (default)

Retained Host and Service Attribute Masks

Format:	**retained_host_attribute_mask=<number>** **retained_service_attribute_mask=<number>**
Example:	**retained_host_attribute_mask=0** **retained_service_attribute_mask=0**

WARNING: This is an advanced feature. You'll need to read the Nagios source code to use this option effectively.

These options determine which host or service attributes are NOT retained across program restarts. The values for these options are a bitwise AND of values specified by the "MODATTR_" definitions in the include/common.h source code file. By default, all host and service attributes are retained.

Retained Process Attribute Masks

Format:	**retained_process_host_attribute_mask=<number>** **retained_process_service_attribute_mask=<number>**
Example:	**retained_process_host_attribute_mask=0** **retained_process_service_attribute_mask=0**

WARNING: This is an advanced feature. You'll need to read the Nagios source code to use this option effectively.

These options determine which process attributes are NOT retained across program restarts. There are two masks because there are often separate host and service process

attributes that can be changed. For example, host checks can be disabled at the program level, while service checks are still enabled. The values for these options are a bitwise AND of values specified by the "MODATTR_" definitions in the include/common.h source code file. By default, all process attributes are retained.

Retained Contact Attribute Masks

Format:	**retained_contact_host_attribute_mask=<number>** **retained_contact_service_attribute_mask=<number>**
Example:	**retained_contact_host_attribute_mask=0** **retained_contact_service_attribute_mask=0**

WARNING: This is an advanced feature. You'll need to read the Nagios source code to use this option effectively.

These options determine which contact attributes are NOT retained across program restarts. There are two masks because there are often separate host and service contact attributes that can be changed. The values for these options are a bitwise AND of values specified by the "MODATTR_" definitions in the include/common.h source code file. By default, all process attributes are retained.

Syslog Logging Option

Format:	**use_syslog=<0/1>**
Example:	**use_syslog=1**

This variable determines whether messages are logged to the syslog facility on your local host. Values are as follows:

- 0 = Don't use syslog facility
- 1 = Use syslog facility

Notification Logging Option

Format:	**log_notifications=<0/1>**
Example:	**log_notifications=1**

This variable determines whether or not notification messages are logged. If you have a lot of contacts or regular service failures your log file will grow relatively quickly. Use this option to keep contact notifications from being logged.

- 0 = Don't log notifications
- 1 = Log notifications

Service Check Retry Logging Option

Format:	**log_service_retries=<0/1>**
Example:	**log_service_retries=1**

This variable determines whether or not service check retries are logged. Service check retries occur when a service check results in a non-OK state, but you have configured Nagios to retry the service more than once before responding to the error. Services in this situation are considered to be in "soft" states. Logging service check retries is mostly useful when attempting to debug Nagios or test out service event handlers.

- 0 = Don't log service check retries
- 1 = Log service check retries

Host Check Retry Logging Option

Format:	**log_host_retries=<0/1>**
Example:	**log_host_retries=1**

This variable determines whether or not host check retries are logged. Logging host check retries is mostly useful when attempting to debug Nagios or test out host event handlers.

- 0 = Don't log host check retries
- 1 = Log host check retries

Event Handler Logging Option

Format:	**log_event_handlers=<0/1>**
Example:	**log_event_handlers=1**

This variable determines whether or not service and host event handlers are logged. Event handlers are optional commands that can be run whenever a service or hosts changes state. Logging event handlers is most useful when debugging Nagios or first trying out your event handler scripts.

- 0 = Don't log event handlers
- 1 = Log event handlers

Initial States Logging Option

Format:	**log_initial_states=<0/1>**
Example:	**log_initial_states=1**

This variable determines whether or not Nagios will force all initial host and service states to be logged, even if they result in an OK state. Initial service and host states are normally only logged when there is a problem on the first check. Enabling this option is useful if you are using an application that scans the log file to determine long-term state statistics for services and hosts.

- 0 = Don't log initial states (default)
- 1 = Log initial states

External Command Logging Option

Format:	**log_external_commands=<0/1>**
Example:	**log_external_commands=1**

This variable determines whether or not Nagios will log external commands that it receives from the external command file. Note: This option does not control whether or not passive service checks (which are a type of external command) get logged. To enable or disable logging of passive checks, use the log_passive_checks option.

- 0 = Don't log external commands
- 1 = Log external commands (default)

Passive Check Logging Option

Format:	**log_passive_checks=<0/1>**
Example:	**log_passive_checks=1**

This variable determines whether or not Nagios will log passive host and service checks that it receives from the external command file. If you are setting up a distributed monitoring environment or plan on handling a large number of passive checks on a regular basis, you may wish to disable this option so your log file doesn't get too large.

- 0 = Don't log passive checks
- 1 = Log passive checks (default)

Global Host Event Handler Option

Format:	**global_host_event_handler=<command>**
Example:	**global_host_event_handler=log-host-event-to-db**

This option allows you to specify a host event handler command that is to be run for every host state change. The global event handler is executed immediately prior to the event handler that you have optionally specified in each host definition. The *command* argument is the short name of a command that you define in your object configuration file. The maximum amount of time that this command can run is controlled by the event_handler_timeout option. More information on event handlers can be found here.

Global Service Event Handler Option

Format:	**global_service_event_handler=<command>**
Example:	**global_service_event_handler=log-service-event-to-db**

This option allows you to specify a service event handler command that is to be run for every service state change. The global event handler is executed immediately prior to the event handler that you have optionally specified in each service definition. The *command* argument is the short name of a command that you define in your object configuration file. The maximum amount of time that this command can run is controlled by the event_handler_timeout option. More information on event handlers can be found here.

Inter-Check Sleep Time

Format:	**sleep_time=<seconds>**
Example:	**sleep_time=1**

This is the number of seconds that Nagios will sleep before checking to see if the next service or host check in the scheduling queue should be executed. Note that Nagios will only sleep after it "catches up" with queued service checks that have fallen behind.

Service Inter-Check Delay Method

Format:	**service_inter_check_delay_method=<n/d/s/x.xx>**
Example:	**service_inter_check_delay_method=s**

This option allows you to control how service checks are initially "spread out" in the event queue. Using a "smart" delay calculation (the default) will cause Nagios to calculate an average check interval and spread initial checks of all services out over that interval, thereby helping to eliminate CPU load spikes. Using no delay is generally *not* recommended, as it will cause all service checks to be scheduled for execution at the same time. This means that you will generally have large CPU spikes when the services are all executed in parallel. More information on how to estimate how the inter-check delay affects service check scheduling can be found here. Values are as follows:

- n = Don't use any delay - schedule all service checks to run immediately (i.e. at the same time!)
- d = Use a "dumb" delay of 1 second between service checks
- s = Use a "smart" delay calculation to spread service checks out evenly (default)
- x.xx = Use a user-supplied inter-check delay of x.xx seconds

Maximum Service Check Spread

Format:	**max_service_check_spread=<minutes>**
Example:	**max_service_check_spread=30**

This option determines the maximum number of minutes from when Nagios starts that all services (that are scheduled to be regularly checked) are checked. This option will automatically adjust the service inter-check delay method (if necessary) to ensure that the initial checks of all services occur within the timeframe you specify. In general, this option will not have an affect on service check scheduling if scheduling information is being retained using the use_retained_scheduling_info option. Default value is **30** (minutes).

Service Interleave Factor

Format:	**service_interleave_factor=<s|*x*>**
Example:	**service_interleave_factor=s**

This variable determines how service checks are interleaved. Interleaving allows for a more even distribution of service checks, reduced load on remote hosts, and faster overall detection of host problems. Setting this value to 1 is equivalent to not interleaving the service checks (this is how versions of Nagios previous to 0.0.5 worked). Set this value to **s** (smart) for automatic calculation of the interleave factor unless you have a specific reason to change it. The best way to understand how interleaving works is to watch the status CGI (detailed view) when Nagios is just starting. You should see that the service check results are spread out as they begin to appear. More information on how interleaving works can be found here.

- *x* = A number greater than or equal to 1 that specifies the interleave factor to use. An interleave factor of 1 is equivalent to not interleaving the service checks.
- s = Use a "smart" interleave factor calculation (default)

Maximum Concurrent Service Checks

Format:	**max_concurrent_checks=<max_checks>**
Example:	**max_concurrent_checks=20**

This option allows you to specify the maximum number of service checks that can be run in parallel at any given time. Specifying a value of 1 for this variable essentially prevents any service checks from being run in parallel. Specifying a value of 0 (the default) does not place any restrictions on the number of concurrent checks. You'll have to modify this value based on the system resources you have available on the machine that runs Nagios, as it directly affects the maximum load that will be imposed on the system (processor utilization, memory, etc.). More information on how to estimate how many concurrent checks you should allow can be found here.

Check Result Reaper Frequency

Format:	**check_result_reaper_frequency=<frequency_in_seconds>**
Example:	**check_result_reaper_frequency=5**

This option allows you to control the frequency *in seconds* of check result "reaper" events. "Reaper" events process the results from host and service checks that have finished executing. These events consitute the core of the monitoring logic in Nagios.

Maximum Check Result Reaper Time

Format:	**max_check_result_reaper_time=<seconds>**
Example:	**max_check_result_reaper_time=30**

This option allows you to control the maximum amount of time *in seconds* that host and service check result "reaper" events are allowed to run. "Reaper" events process the results from host and service checks that have finished executing. If there are a lot of results to process, reaper events may take a long time to finish, which might delay timely execution of new host and service checks. This variable allows you to limit the amount of time that an individual reaper event will run before it hands control back over to Nagios for other portions of the monitoring logic.

Check Result Path

Format:	**check_result_path=<path>**
Example:	**check_result_path=/var/spool/nagios/checkresults**

This options determines which directory Nagios will use to temporarily store host and service check results before they are processed. This directory should not be used to store any other files, as Nagios will periodically clean this directory of old file (see the max_check_result_file_age option for more information).

Note: Make sure that only a single instance of Nagios has access to the check result path. If multiple instances of Nagios have their check result path set to the same directory, you will run into problems with check results being processed (incorrectly) by the wrong instance of Nagios!

Max Check Result File Age

Format:	**max_check_result_file_age=<seconds>**
Example:	**max_check_result_file_age=3600**

This options determines the maximum age in seconds that Nagios will consider check result files found in the check_result_path directory to be valid. Check result files that are older that this threshold will be deleted by Nagios and the check results they contain will not be processed. By using a value of zero (0) with this option, Nagios will process all check result files - even if they're older than your hardware :-).

Host Inter-Check Delay Method

Format:	**host_inter_check_delay_method=<n/d/s/x.xx>**
Example:	**host_inter_check_delay_method=s**

This option allows you to control how host checks *that are scheduled to be checked on a regular basis* are initially "spread out" in the event queue. Using a "smart" delay calculation (the default) will cause Nagios to calculate an average check interval and spread initial checks of all hosts out over that interval, thereby helping to eliminate CPU load spikes. Using no delay is generally *not* recommended. Using no delay will cause all host checks to be scheduled for execution at the same time. More information on how to estimate how the inter-check delay affects host check scheduling can be found here.Values are as follows:

- n = Don't use any delay - schedule all host checks to run immediately (i.e. at the same time!)
- d = Use a "dumb" delay of 1 second between host checks
- s = Use a "smart" delay calculation to spread host checks out evenly (default)
- x.xx = Use a user-supplied inter-check delay of x.xx seconds

Maximum Host Check Spread

Format:	**max_host_check_spread=<minutes>**
Example:	**max_host_check_spread=30**

This option determines the maximum number of minutes from when Nagios starts that all hosts (that are scheduled to be regularly checked) are checked. This option will automatically adjust the host inter-check delay method (if necessary) to ensure that the initial checks of all hosts occur within the timeframe you specify. In general, this option will not have an affect on host check scheduling if scheduling information is being retained using the use_retained_scheduling_info option. Default value is **30** (minutes).

Timing Interval Length

Format:	**interval_length=<seconds>**
Example:	**interval_length=60**

This is the number of seconds per "unit interval" used for timing in the scheduling queue, re-notifications, etc. "Units intervals" are used in the object configuration file to determine how often to run a service check, how often to re-notify a contact, etc.

Important: The default value for this is set to 60, which means that a "unit value" of 1 in the object configuration file will mean 60 seconds (1 minute). I have not really tested other values for this variable, so proceed at your own risk if you decide to do so!

Auto-Rescheduling Option

Format:	**auto_reschedule_checks=<0/1>**
Example:	**auto_reschedule_checks=1**

This option determines whether or not Nagios will attempt to automatically reschedule active host and service checks to "smooth" them out over time. This can help to balance the load on the monitoring server, as it will attempt to keep the time between consecutive checks consistent, at the expense of executing checks on a more rigid schedule.

WARNING: THIS IS AN EXPERIMENTAL FEATURE AND MAY BE REMOVED IN FUTURE VERSIONS. ENABLING THIS OPTION CAN DEGRADE PERFORMANCE - RATHER THAN INCREASE IT - IF USED IMPROPERLY!

Auto-Rescheduling Interval

Format:	**auto_rescheduling_interval=<seconds>**
Example:	**auto_rescheduling_interval=30**

This option determines how often (in seconds) Nagios will attempt to automatically reschedule checks. This option only has an effect if the auto_reschedule_checks option is enabled. Default is 30 seconds.

WARNING: THIS IS AN EXPERIMENTAL FEATURE AND MAY BE REMOVED IN FUTURE VERSIONS. ENABLING THE AUTO-RESCHEDULING OPTION CAN DEGRADE PERFORMANCE - RATHER THAN INCREASE IT - IF USED IMPROPERLY!

Auto-Rescheduling Window

Format:	**auto_rescheduling_window=<seconds>**
Example:	**auto_rescheduling_window=180**

This option determines the "window" of time (in seconds) that Nagios will look at when automatically rescheduling checks. Only host and service checks that occur in the next X seconds (determined by this variable) will be rescheduled. This option only has an effect if the auto_reschedule_checks option is enabled. Default is 180 seconds (3 minutes).

WARNING: THIS IS AN EXPERIMENTAL FEATURE AND MAY BE REMOVED IN FUTURE VERSIONS. ENABLING THE AUTO-RESCHEDULING OPTION CAN DEGRADE PERFORMANCE - RATHER THAN INCREASE IT - IF USED IMPROPERLY!

Aggressive Host Checking Option

Format:	**use_aggressive_host_checking=<0/1>**
Example:	**use_aggressive_host_checking=0**

Nagios tries to be smart about how and when it checks the status of hosts. In general, disabling this option will allow Nagios to make some smarter decisions and check hosts a bit faster. Enabling this option will increase the amount of time required to check hosts, but may improve reliability a bit. Unless you have problems with Nagios not recognizing that a host recovered, I would suggest **not** enabling this option.

- 0 = Don't use aggressive host checking (default)
- 1 = Use aggressive host checking

Translate Passive Host Checks Option

Format:	**translate_passive_host_checks=<0/1>**
Example:	**translate_passive_host_checks=1**

This option determines whether or not Nagios will translate DOWN/UNREACHABLE passive host check results to their "correct" state from the viewpoint of the local Nagios instance. This can be very useful in distributed and failover monitoring installations. More information on passive check state translation can be found here.

- 0 = Disable check translation (default)
- 1 = Enable check translation

Passive Host Checks Are SOFT Option

Format:	**passive_host_checks_are_soft=<0/1>**
Example:	**passive_host_checks_are_soft=1**

This option determines whether or not Nagios will treat passive host checks as HARD states or SOFT states. By default, a passive host check result will put a host into a HARD state type. You can change this behavior by enabling this option.

- 0 = Passive host checks are HARD (default)
- 1 = Passive host checks are SOFT

Predictive Host Dependency Checks Option

Format:	**enable_predictive_host_dependency_checks=<0/1>**
Example:	**enable_predictive_host_dependency_checks=1**

This option determines whether or not Nagios will execute predictive checks of hosts that are being depended upon (as defined in host dependencies) for a particular host when it changes state. Predictive checks help ensure that the dependency logic is as accurate as possible. More information on how predictive checks work can be found here.

- 0 = Disable predictive checks
- 1 = Enable predictive checks (default)

Predictive Service Dependency Checks Option

Format:	**enable_predictive_service_dependency_checks=<0/1>**
Example:	**enable_predictive_service_dependency_checks=1**

This option determines whether or not Nagios will execute predictive checks of services that are being depended upon (as defined in service dependencies) for a particular service when it changes state. Predictive checks help ensure that the dependency logic is as accurate as possible. More information on how predictive checks work can be found here.

- 0 = Disable predictive checks
- 1 = Enable predictive checks (default)

Cached Host Check Horizon

Format:	**cached_host_check_horizon=<seconds>**
Example:	**cached_host_check_horizon=15**

This option determines the maximum amount of time (in seconds) that the state of a previous host check is considered current. Cached host states (from host checks that were performed more recently than the time specified by this value) can improve host check performance immensely. Too high of a value for this option may result in (temporarily) inaccurate host states, while a low value may result in a performance hit for host checks. Use a value of 0 if you want to disable host check caching. More information on cached checks can be found here.

Cached Service Check Horizon

Format:	**cached_service_check_horizon=<seconds>**
Example:	**cached_service_check_horizon=15**

This option determines the maximum amount of time (in seconds) that the state of a previous service check is considered current. Cached service states (from service checks that were performed more recently than the time specified by this value) can improve service check performance when a lot of service dependencies are used. Too high of a value for this option may result in inaccuracies in the service dependency logic. Use a value of 0 if you want to disable service check caching. More information on cached checks can be found here.

Large Installation Tweaks Option

Format:	**use_large_installation_tweaks=<0/1>**
Example:	**use_large_installation_tweaks=0**

This option determines whether or not the Nagios daemon will take several shortcuts to improve performance. These shortcuts result in the loss of a few features, but larger installations will likely see a lot of benefit from doing so. More information on what optimizations are taken when you enable this option can be found here.

- 0 = Don't use tweaks (default)
- 1 = Use tweaks

Child Process Memory Option

Format:	**free_child_process_memory=<0/1>**
Example:	**free_child_process_memory=0**

This option determines whether or not Nagios will free memory in child processes when they are fork()ed off from the main process. By default, Nagios frees memory. However, if the use_large_installation_tweaks option is enabled, it will not. By defining this option in your configuration file, you are able to override things to get the behavior you want.

- 0 = Don't free memory

- 1 = Free memory

Child Processes Fork Twice

Format:	**child_processes_fork_twice=<0/1>**
Example:	**child_processes_fork_twice=0**

This option determines whether or not Nagios will fork() child processes twice when it executes host and service checks. By default, Nagios fork()s twice. However, if the use_large_installation_tweaks option is enabled, it will only fork() once. By defining this option in your configuration file, you are able to override things to get the behavior you want.

- 0 = Fork() just once
- 1 = Fork() twice

Environment Macros Option

Format:	**enable_environment_macros=<0/1>**
Example:	**enable_environment_macros=0**

This option determines whether or not the Nagios daemon will make all standard macros available as environment variables to your check, notification, event hander, etc. commands. In large Nagios installations this can be problematic because it takes additional memory and (more importantly) CPU to compute the values of all macros and make them available to the environment.

- 0 = Don't make macros available as environment variables
- 1 = Make macros available as environment variables (default)

Flap Detection Option

Format:	**enable_flap_detection=<0/1>**
Example:	**enable_flap_detection=0**

This option determines whether or not Nagios will try and detect hosts and services that are "flapping". Flapping occurs when a host or service changes between states too frequently, resulting in a barrage of notifications being sent out. When Nagios detects that a host or service is flapping, it will temporarily suppress notifications for that host/service until it stops flapping. Flap detection is very experimental at this point, so use this feature with caution! More information on how flap detection and handling works can be found here. Note: If you have state retention enabled, Nagios will ignore this setting when it (re)starts and use the last known setting for this option (as stored in the state retention file), *unless* you disable the use_retained_program_state option. If you want to change this option when state retention is active (and the use_retained_program_state is enabled), you'll have to use the appropriate external command or change it via the web interface.

- 0 = Don't enable flap detection (default)
- 1 = Enable flap detection

Low Service Flap Threshold

Format:	**low_service_flap_threshold=<percent>**
Example:	**low_service_flap_threshold=25.0**

This option is used to set the low threshold for detection of service flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

High Service Flap Threshold

Format:	**high_service_flap_threshold=<percent>**
Example:	**high_service_flap_threshold=50.0**

This option is used to set the high threshold for detection of service flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

Low Host Flap Threshold

Format:	**low_host_flap_threshold=<percent>**
Example:	**low_host_flap_threshold=25.0**

This option is used to set the low threshold for detection of host flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

High Host Flap Threshold

Format:	**high_host_flap_threshold=<percent>**
Example:	**high_host_flap_threshold=50.0**

This option is used to set the high threshold for detection of host flapping. For more information on how flap detection and handling works (and how this option affects things) read this.

Soft State Dependencies Option

Format:	**soft_state_dependencies=<0/1>**
Example:	**soft_state_dependencies=0**

This option determines whether or not Nagios will use soft state information when checking host and service dependencies. Normally Nagios will only use the latest hard host or service state when checking dependencies. If you want it to use the latest state (regardless of whether its a soft or hard state type), enable this option.

- 0 = Don't use soft state dependencies (default)
- 1 = Use soft state dependencies

Service Check Timeout

Format:	**service_check_timeout=<seconds>**
Example:	**service_check_timeout=60**

This is the maximum number of seconds that Nagios will allow service checks to run. If checks exceed this limit, they are killed and a CRITICAL state is returned. A timeout error will also be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off plugins which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each service check normally finishes executing within this time limit. If a service check runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Host Check Timeout

Format:	**host_check_timeout=<seconds>**
Example:	**host_check_timeout=60**

This is the maximum number of seconds that Nagios will allow host checks to run. If checks exceed this limit, they are killed and a CRITICAL state is returned and the host will be assumed to be DOWN. A timeout error will also be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off plugins which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each host check normally finishes executing within this time limit. If a host check runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Event Handler Timeout

Format:	**event_handler_timeout=<seconds>**
Example:	**event_handler_timeout=60**

This is the maximum number of seconds that Nagios will allow event handlers to be run. If an event handler exceeds this time limit it will be killed and a warning will be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off commands which are misbehaving and not exiting in a timely manner. It should be set to something high (like 60 seconds or more), so that each event handler command normally finishes executing within this time limit. If an event handler runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Notification Timeout

Format:	**notification_timeout=<seconds>**
Example:	**notification_timeout=60**

This is the maximum number of seconds that Nagios will allow notification commands to be run. If a notification command exceeds this time limit it will be killed and a warning will be logged.

There is often widespread confusion as to what this option really does. It is meant to be used as a last ditch mechanism to kill off commands which are misbehaving and not exiting in

a timely manner. It should be set to something high (like 60 seconds or more), so that each notification command finishes executing within this time limit. If a notification command runs longer than this limit, Nagios will kill it off thinking it is a runaway processes.

Obsessive Compulsive Service Processor Timeout

Format:	**ocsp_timeout=<seconds>**
Example:	**ocsp_timeout=5**

This is the maximum number of seconds that Nagios will allow an obsessive compulsive service processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

Obsessive Compulsive Host Processor Timeout

Format:	**ochp_timeout=<seconds>**
Example:	**ochp_timeout=5**

This is the maximum number of seconds that Nagios will allow an obsessive compulsive host processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

Performance Data Processor Command Timeout

Format:	**perfdata_timeout=<seconds>**
Example:	**perfdata_timeout=5**

This is the maximum number of seconds that Nagios will allow a host performance data processor command or service performance data processor command to be run. If a command exceeds this time limit it will be killed and a warning will be logged.

Obsess Over Services Option

Format:	**obsess_over_services=<0/1>**
Example:	**obsess_over_services=1**

This value determines whether or not Nagios will "obsess" over service checks results and run the obsessive compulsive service processor command you define. I know - funny name, but it was all I could think of. This option is useful for performing distributed monitoring. If you're not doing distributed monitoring, don't enable this option.

- 0 = Don't obsess over services (default)
- 1 = Obsess over services

Obsessive Compulsive Service Processor Command

Format:	**ocsp_command=<command>**
Example:	**ocsp_command=obsessive_service_handler**

This option allows you to specify a command to be run after *every* service check, which can be useful in distributed monitoring. This command is executed after any event handler or notification commands. The *command* argument is the short name of a command definition that you define in your object configuration file. The maximum amount of time that this command can run is controlled by the ocsp_timeout option. More information on distributed monitoring can be found here. This command is only executed if the obsess_over_services option is enabled globally and if the *obsess_over_service* directive in the service definition is enabled.

Obsess Over Hosts Option

Format:	**obsess_over_hosts=<0/1>**
Example:	**obsess_over_hosts=1**

This value determines whether or not Nagios will "obsess" over host checks results and run the obsessive compulsive host processor command you define. I know - funny name, but it was all I could think of. This option is useful for performing distributed monitoring. If you're not doing distributed monitoring, don't enable this option.

- 0 = Don't obsess over hosts (default)
- 1 = Obsess over hosts

Obsessive Compulsive Host Processor Command

Format:	**ochp_command=<command>**
Example:	**ochp_command=obsessive_host_handler**

This option allows you to specify a command to be run after *every* host check, which can be useful in distributed monitoring. This command is executed after any event handler or notification commands. The *command* argument is the short name of a command definition that you define in your object configuration file. The maximum amount of time that this command can run is controlled by the ochp_timeout option. More information on distributed monitoring can be found here. This command is only executed if the obsess_over_hosts option is enabled globally and if the *obsess_over_host* directive in the host definition is enabled.

Performance Data Processing Option

Format:	**process_performance_data=<0/1>**
Example:	**process_performance_data=1**

This value determines whether or not Nagios will process host and service check performance data.

- 0 = Don't process performance data (default)
- 1 = Process performance data

Host Performance Data Processing Command

Format:	**host_perfdata_command=<command>**
Example:	**host_perfdata_command=process-host-perfdata**

This option allows you to specify a command to be run after *every* host check to process host performance data that may be returned from the check. The *command* argument is the short name of a command definition that you define in your object configuration file. This command is only executed if the process_performance_data option is enabled globally and if the *process_perf_data* directive in the host definition is enabled.

Service Performance Data Processing Command

Format:	**service_perfdata_command=<command>**
Example:	**service_perfdata_command=process-service-perfdata**

This option allows you to specify a command to be run after *every* service check to process service performance data that may be returned from the check. The *command* argument is the short name of a command definition that you define in your object configuration file. This command is only executed if the process_performance_data option is enabled globally and if the *process_perf_data* directive in the service definition is enabled.

Host Performance Data File

Format:	**host_perfdata_file=<file_name>**
Example:	**host_perfdata_file=/usr/local/nagios/var/host-perfdata.dat**

This option allows you to specify a file to which host performance data will be written after every host check. Data will be written to the performance file as specified by the host_perfdata_file_template option. Performance data is only written to this file if the process_performance_data option is enabled globally and if the *process_perf_data* directive in the host definition is enabled.

Service Performance Data File

Format:	**service_perfdata_file=<file_name>**
Example:	**service_perfdata_file=/usr/local/nagios/var/service-perfdata.dat**

This option allows you to specify a file to which service performance data will be written after every service check. Data will be written to the performance file as specified by the service_perfdata_file_template option. Performance data is only written to this file if the process_performance_data option is enabled globally and if the *process_perf_data* directive in the service definition is enabled.

Host Performance Data File Template

Format:	**host_perfdata_file_template=<template>**
Example:	**host_perfdata_file_template=[HOSTPERFDATA]\t$TIMET$\t$HOSTNAME$\t$HOSTEXECUTIONTIME$\t$HOSTOUTPUT$\t$HOSTPERFDATA$**

This option determines what (and how) data is written to the host performance data file. The template may contain macros, special characters (\t for tab, \r for carriage return, \n for newline) and plain text. A newline is automatically added after each write to the performance data file.

Service Performance Data File Template

Format:	**service_perfdata_file_template=<template>**
Example:	**service_perfdata_file_template=[SERVICEPERFDATA]\t$TIMET$\t$HOSTNAME$\t$SERVICEDESC$\t$SERVICEEXECUTIONTIME$\t$SERVICELATENCY$\t$SERVICEOUTPUT$\t$SERVICEPERFDATA$**

This option determines what (and how) data is written to the service performance data file. The template may contain macros, special characters (\t for tab, \r for carriage return, \n for newline) and plain text. A newline is automatically added after each write to the performance data file.

Host Performance Data File Mode

Format:	**host_perfdata_file_mode=<mode>**
Example:	**host_perfdata_file_mode=a**

This option determines how the host performance data file is opened. Unless the file is a named pipe you'll probably want to use the default mode of append.

- a = Open file in append mode (default)
- w = Open file in write mode
- p = Open in non-blocking read/write mode (useful when writing to pipes)

Service Performance Data File Mode

Format:	**service_perfdata_file_mode=<mode>**
Example:	**service_perfdata_file_mode=a**

This option determines how the service performance data file is opened. Unless the file is a named pipe you'll probably want to use the default mode of append.

- a = Open file in append mode (default)
- w = Open file in write mode
- p = Open in non-blocking read/write mode (useful when writing to pipes)

Host Performance Data File Processing Interval

Format:	**host_perfdata_file_processing_interval=<seconds>**
Example:	**host_perfdata_file_processing_interval=0**

This option allows you to specify the interval (in seconds) at which the host performance data file is processed using the host performance data file processing command. A value of 0 indicates that the performance data file should not be processed at regular intervals.

Service Performance Data File Processing Interval

Format:	**service_perfdata_file_processing_interval=<seconds>**
Example:	**service_perfdata_file_processing_interval=0**

This option allows you to specify the interval (in seconds) at which the service performance data file is processed using the service performance data file processing command. A value of 0 indicates that the performance data file should not be processed at regular intervals.

Host Performance Data File Processing Command

Format:	**host_perfdata_file_processing_command=<command>**
Example:	**host_perfdata_file_processing_command=process-host-perfdata-file**

This option allows you to specify the command that should be executed to process the host performance data file. The *command* argument is the short name of a command definition that you define in your object configuration file. The interval at which this command is executed is determined by the host_perfdata_file_processing_interval directive.

Service Performance Data File Processing Command

Format:	**service_perfdata_file_processing_command=<command>**
Example:	**service_perfdata_file_processing_command=process-service-perfdata-file**

This option allows you to specify the command that should be executed to process the service performance data file. The *command* argument is the short name of a command definition that you define in your object configuration file. The interval at which this command is executed is determined by the service_perfdata_file_processing_interval directive.

Orphaned Service Check Option

Format:	**check_for_orphaned_services=<0/1>**
Example:	**check_for_orphaned_services=1**

This option allows you to enable or disable checks for orphaned service checks. Orphaned service checks are checks which have been executed and have been removed from the event queue, but have not had any results reported in a long time. Since no results have come back in for the service, it is not rescheduled in the event queue. This can cause service checks to stop being executed. Normally it is very rare for this to happen - it might happen if an external user or process killed off the process that was being used to execute a service check. If this option is enabled and Nagios finds that results for a particular service check have not come back, it will log an error message and reschedule the service check. If you start seeing service checks that never seem to get rescheduled, enable this option and see if you notice any log messages about orphaned services.

- 0 = Don't check for orphaned service checks
- 1 = Check for orphaned service checks (default)

Orphaned Host Check Option

Format:	**check_for_orphaned_hosts=<0/1>**
Example:	**check_for_orphaned_hosts=1**

This option allows you to enable or disable checks for orphaned hoste checks. Orphaned host checks are checks which have been executed and have been removed from the event queue, but have not had any results reported in a long time. Since no results have come back in for the host, it is not rescheduled in the event queue. This can cause host checks to stop being executed. Normally it is very rare for this to happen - it might happen if an external user or process killed off the process that was being used to execute a host check. If this option is enabled and Nagios finds that results for a particular host check have not come back, it will log an error message and reschedule the host check. If you start seeing host checks that never seem to get rescheduled, enable this option and see if you notice any log messages about orphaned hosts.

- 0 = Don't check for orphaned host checks
- 1 = Check for orphaned host checks (default)

Service Freshness Checking Option

Format:	**check_service_freshness=<0/1>**
Example:	**check_service_freshness=0**

This option determines whether or not Nagios will periodically check the "freshness" of service checks. Enabling this option is useful for helping to ensure that passive service checks are received in a timely manner. More information on freshness checking can be found here.

- 0 = Don't check service freshness
- 1 = Check service freshness (default)

Service Freshness Check Interval

Format:	**service_freshness_check_interval=<seconds>**
Example:	**service_freshness_check_interval=60**

This setting determines how often (in seconds) Nagios will periodically check the "freshness" of service check results. If you have disabled service freshness checking (with the check_service_freshness option), this option has no effect. More information on freshness checking can be found here.

Host Freshness Checking Option

Format:	**check_host_freshness=<0/1>**
Example:	**check_host_freshness=0**

This option determines whether or not Nagios will periodically check the "freshness" of host checks. Enabling this option is useful for helping to ensure that passive host checks are received in a timely manner. More information on freshness checking can be found here.

- 0 = Don't check host freshness
- 1 = Check host freshness (default)

Host Freshness Check Interval

Format:	**host_freshness_check_interval=<seconds>**
Example:	**host_freshness_check_interval=60**

This setting determines how often (in seconds) Nagios will periodically check the "freshness" of host check results. If you have disabled host freshness checking (with the check_host_freshness option), this option has no effect. More information on freshness checking can be found here.

Additional Freshness Threshold Latency Option

Format:	**additional_freshness_latency=<#>**
Example:	**additional_freshness_latency=15**

This option determines the number of seconds Nagios will add to any host or services freshness threshold it automatically calculates (e.g. those not specified explicity by the user). More information on freshness checking can be found here.

Embedded Perl Interpreter Option

Format:	**enable_embedded_perl=<0/1>**
Example:	**enable_embedded_perl=1**

This setting determines whether or not the embedded Perl interpreter is enabled on a program-wide basis. Nagios must be compiled with support for embedded Perl for this option to have an effect. More information on the embedded Perl interpreter can be found here.

Embedded Perl Implicit Use Option

Format:	**use_embedded_perl_implicitly=<0/1>**
Example:	**use_embedded_perl_implicitly=1**

This setting determines whether or not the embedded Perl interpreter should be used for Perl plugins/scripts that do not explicitly enable/disable it. Nagios must be compiled with support for embedded Perl for this option to have an effect. More information on the embedded Perl interpreter and the effect of this setting can be found here.

Date Format

Format:	**date_format=<option>**
Example:	**date_format=us**

This option allows you to specify what kind of date/time format Nagios should use in the web interface and date/time macros. Possible options (along with example output) include:

Option	Output Format	Sample Output
us	MM/DD/YYYY HH:MM:SS	06/30/2002 03:15:00
euro	DD/MM/YYYY HH:MM:SS	30/06/2002 03:15:00
iso8601	YYYY-MM-DD HH:MM:SS	2002-06-30 03:15:00
strict-iso8601	YYYY-MM-DDTHH:MM:SS	2002-06-30T03:15:00

Timezone Option

Format:	**use_timezone=<tz>**
Example:	**use_timezone=US/Mountain**

This option allows you to override the default timezone that this instance of Nagios runs in. Useful if you have multiple instances of Nagios that need to run from the same server, but have different local times associated with them. If not specified, Nagios will use the system configured timezone.

Note: If you use this option to specify a custom timezone, you will also need to alter the Apache configuration directives for the CGIs to specify the timezone you want. Example:

```
<Directory "/usr/local/nagios/sbin/">
SetEnv TZ "US/Mountain"
...
</Directory>
```

Illegal Object Name Characters

Format:	**illegal_object_name_chars=<chars...>**
Example:	**illegal_object_name_chars=~!$%^&*"|'<>?,()=**

This option allows you to specify illegal characters that cannot be used in host names, service descriptions, or names of other object types. Nagios will allow you to use most characters in object definitions, but I recommend not using the characters shown in the example above. Doing may give you problems in the web interface, notification commands, etc.

Illegal Macro Output Characters

Format:	**illegal_macro_output_chars=<chars...>**
Example:	**illegal_macro_output_chars=~$^&"|'<>**

This option allows you to specify illegal characters that should be stripped from macros before being used in notifications, event handlers, and other commands. This DOES NOT affect macros used in service or host check commands. You can choose to not strip out the characters shown in the example above, but I recommend you do not do this. Some of these characters are interpreted by the shell (i.e. the backtick) and can lead to security problems. The following macros are stripped of the characters you specify:

$HOSTOUTPUT$, **$HOSTPERFDATA$**, **$HOSTACKAUTHOR$**, **$HOSTACKCOMMENT$**, **$SERVICEOUTPUT$**, **$SERVICEPERFDATA$**, **$SERVICEACKAUTHOR$**, and **$SERVICEACKCOMMENT$**

Regular Expression Matching Option

Format:	**use_regexp_matching=<0/1>**
Example:	**use_regexp_matching=0**

This option determines whether or not various directives in your object definitions will be processed as regular expressions. More information on how this works can be found here.

- 0 = Don't use regular expression matching (default)
- 1 = Use regular expression matching

True Regular Expression Matching Option

Format:	**use_true_regexp_matching=<0/1>**
Example:	**use_true_regexp_matching=0**

If you've enabled regular expression matching of various object directives using the use_regexp_matching option, this option will determine when object directives are treated as regular expressions. If this option is disabled (the default), directives will only be treated as regular expressions if they contain *****, **?**, **+**, or **\\.** If this option is enabled, all appropriate directives will be treated as regular expression - be careful when enabling this! More information on how this works can be found here.

- 0 = Don't use true regular expression matching (default)
- 1 = Use true regular expression matching

Administrator Email Address

Format:	**admin_email=<email_address>**
Example:	**admin_email=root@localhost.localdomain**

This is the email address for the administrator of the local machine (i.e. the one that Nagios is running on). This value can be used in notification commands by using the **$ADMINEMAIL$** macro.

Administrator Pager

Format:	**admin_pager=<pager_number_or_pager_email_gateway>**
Example:	**admin_pager=pageroot@localhost.localdomain**

This is the pager number (or pager email gateway) for the administrator of the local machine (i.e. the one that Nagios is running on). The pager number/address can be used in notification commands by using the **$ADMINPAGER$** macro.

Event Broker Options

Format:	**event_broker_options=<#>**
Example:	**event_broker_options=-1**

This option controls what (if any) data gets sent to the event broker and, in turn, to any loaded event broker modules. This is an advanced option. When in doubt, either broker nothing (if not using event broker modules) or broker everything (if using event broker modules). Possible values are shown below.

- 0 = Broker nothing
- -1 = Broker everything
- # = See BROKER_* definitions in source code (include/broker.h) for other values that can be OR'ed together

Event Broker Modules

Format:	**broker_module=<modulepath> [moduleargs]**
Example:	**broker_module=/usr/local/nagios/bin/ndomod.o cfg_file=/usr/local/nagios/etc/ndomod.cfg**

This directive is used to specify an event broker module that should by loaded by Nagios at startup. Use multiple directives if you want to load more than one module. Arguments that should be passed to the module at startup are seperated from the module path by a space.

!!! WARNING !!!

Do NOT overwrite modules while they are being used by Nagios or Nagios will crash in a fiery display of SEGFAULT glory. This is a bug/limitation either in dlopen(), the kernel, and/or the filesystem. And maybe Nagios...

The correct/safe way of updating a module is by using one of these methods:

1. Shutdown Nagios, replace the module file, restart Nagios
2. While Nagios is running... delete the original module file, move the new module file into place, restart Nagios

Debug File

Format:	**debug_file=<file_name>**
Example:	**debug_file=/usr/local/nagios/var/nagios.debug**

This option determines where Nagios should write debugging information. What (if any) information is written is determined by the debug_level and debug_verbosity options. You can have Nagios automaticaly rotate the debug file when it reaches a certain size by using the max_debug_file_size option.

Debug Level

Format:	**debug_level=<#>**
Example:	**debug_level=24**

This option determines what type of information Nagios should write to the debug_file. This value is a logical OR of the values below.

- -1 = Log everything
- 0 = Log nothing (default)
- 1 = Function enter/exit information
- 2 = Config information
- 4 = Process information
- 8 = Scheduled event information
- 16 = Host/service check information
- 32 = Notification information
- 64 = Event broker information

Debug Verbosity

Format:	**debug_verbosity=<#>**
Example:	**debug_verbosity=1**

This option determines how much debugging information Nagios should write to the debug_file.

- 0 = Basic information
- 1 = More detailed information (default)
- 2 = Highly detailed information

Maximum Debug File Size

Format:	**max_debug_file_size=<#>**
Example:	**max_debug_file_size=1000000**

This option determines the maximum size (in bytes) of the debug file. If the file grows larger than this size, it will be renamed with a .old extension. If a file already exists with a .old extension it will automatically be deleted. This helps ensure your disk space usage doesn't get out of control when debugging Nagios.

Generated in 0.211 seconds