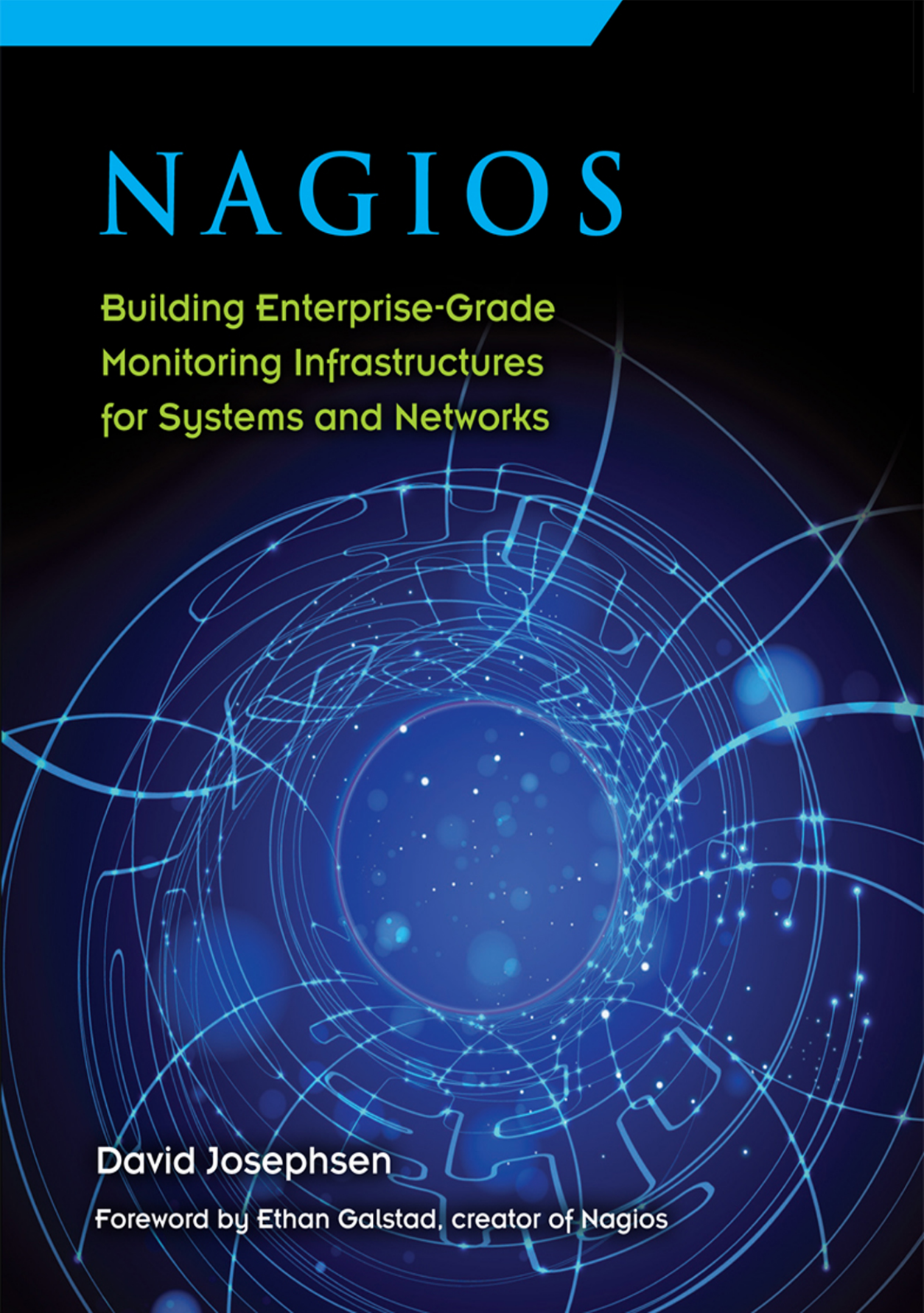
NAGIOS

Building Enterprise-Grade Monitoring Infrastructures for Systems and Networks



David Josephsen

Foreword by Ethan Galstad, creator of Nagios

Nagios

Table 4.4 *Security Related cgi.cfg Directives*

use_authentication	This directive, set to 1 (on) by default, tells the CGIs to use authentication information from the web server. Because the web interface can control how Nagios operates, turning authentication off is a bad idea. If you want everyone to see everything, use default_user_name instead.
default_user_name	Usually set to guest, the default_user_name can be granted permissions that will be inherited by all other users. For example, if jdoe doesn't explicitly have access to see information on a service, but the default user does, jdoe will be able to see the service because all users can see what the default user can. If you want everyone to be able to see all hosts, uncomment this directive and list the default user in authorized_for_all_hosts.
authorized_for_system_information	A comma-separated list of users who are allowed to see information related to the Nagios daemon.
authorized_for_configuration_information	A comma-separated list of users who are allowed to see the contents of the configuration files via the web interface.
authorized_for_system_commands	A comma-separated list of users who are allowed to execute commands relating to the Nagios daemon, such as shutdown and restart.
authorized_for_all_services	A comma-separated list of users who are allowed to see information related to any service that Nagios is monitoring.
authorized_for_all_hosts	Like the above, except for hosts.
authorized_for_all_service_commands	A comma-separated list of users who are allowed to execute commands related to services, such as rescheduling checks and disabling notifications.
authorized_for_all_host_commands	Like the above, except for hosts.

Templates

The brunt of Nagios configuration is made up of object definitions as described in the earlier section "Objects and Definitions." Object definitions involve varying degrees of complexity. Command definitions, for example, are normally composed of no more than two or three lines of text. Service definitions, on the other hand, may contain 31 directives, 11 of which are mandatory. For 100 hosts with 1 service each, that's 1,100 lines of configuration just for service definitions, most of which are redundant. Thankfully, Nagios has a few built-in features that mitigate the need for most of the typing. For example, any of the definitions that

refer to hosts may refer to a list of comma-separated hosts instead. Nagios 2 and later allows you to specify a host group instead of a host for some definitions that refer to hosts. Nagios 3 and later allows you to nest groups in other groups. These three features alone bring our 1,100 lines back to 11.

Another wrist-saving feature is template-based configuration. Templates capture redundant directives inside special definitions. Normal objects can then refer to the template and inherit directives instead of specifying them explicitly. Template definitions look and act exactly like their counterparts with two exceptions: the `register` directive and the `name` directive. Listing 4.4 is a templatized version of the host definition from Listing 4.1.

Listing 4.4 *A Host Template and Consumer Definition*

```
# This is my template
define host{
   name                    hostTemplate
   check_command           check-host-alive
   max_check_attempts      5
   contact_groups          admins
   notification_interval   30
   notification_period     24x7
   notification_options    d,u,r
   register                0
}

# myHost is shorter now that it inherits from hostTemplate
define host{
   host_name    myHost
   alias        My Favorite Host
   address      192.168.1.254
   parents      myotherhost
   use          hostTemplate
}
```

As you can see, both definitions define objects of type `host`. The host template, however, has a `name` directive (instead of a host_name directive) and a `register` directive, which is set to 0. Both `name` and `register` are specific to templates despite the object type, so any other type of template (like a service template) is defined the same way. The `name` directive is self-explanatory; it gives the template a name other objects can refer to. Setting the `register` directive to zero tells Nagios not to treat the object as a host object (don't register it), but rather let other objects inherit settings from it (make it a template).

By adding a `use` directive to myHost's definition, we've instructed Nagios to let myHost inherit settings from the template. The host object will inherit anything that is specified in the template. The host object will override any directives it has in common with the template.

Templates may, in turn, inherit properties from other templates, so it's quite common practice with host definitions to define a global host template, then several OS-specific templates, and then hosts that refer to them. Technically, any object can inherit properties from any other object of the same type, registered or not,[7] to an infinite degree via the `use` directive. I highly recommend the use of template-based configuration; it's very flexible, saves lots of redundant configuration, and makes for readable config files.

Timeperiods

Required for daemon start

Refers to: none

Referred to by: host, service, contact, hostescalation, serviceescalation

Listing 4.5 *Timeperiod Example*

```
define timeperiod{
   timeperiod_name   nonworkhours
   alias             Non-Work Hours
   sunday            00:00-24:00
   monday            00:00-09:00,17:00-24:00
   tuesday           00:00-09:00,17:00-24:00
   wednesday         00:00-09:00,17:00-24:00
   thursday          00:00-09:00,17:00-24:00
   friday            00:00-09:00,17:00-24:00
   saturday          00:00-24:00
}
```

Timeperiods define blocks of time that many other objects reference in the context of operational hours or blackout periods. The timeperiod definition is agnostic; the period of time it defines is not specific to any particular purpose, so two different objects may refer to the same time period for completely different reasons.

Timeperiod definitions have one directive for each day of the week. Omitting a day altogether means the entire day is not included in the time period. Like all other objects, timeperiods may inherit directives from other timeperiods or timeperiod templates. Multiple blocks of time in the same day may be specified by separating them with commas.

In Nagios 3.0, several directives were added to the weekday names to give administrators more flexibility in defining time periods. These are as follows:

Calendar date:

```
2013-01-01         00:00-24:00
```

Specific month date

```
January 1st         00:00-24:00
```

Generic month date

```
Day 15              00:00-24:00
```

Offset weekday of specific month

```
2nd Tuesday in December        00:00-24:00
```

Offset weekday

```
3rd Monday          00:00-24:00
```

Commands

Required for daemon start

Refers to: none

Referred to by: host, service, contact

Listing 4.6 *Command Example*

```
define command{
   command_name    check_ping
   command_line    $USER1$/check_ping –H $HOSTADDRESS$ -w $ARG1$ -c
$ARG2$ -p 5
```

Though composed of only two directives, command definitions are central to the functionality of Nagios. Commands are the means by which Nagios may call external programs and, as we'll see in other definitions throughout the chapter, Nagios calls external programs often.

The most common use of the command object is for calling plug-ins. As previously mentioned, there are only two directives: `command_name`, which gives the object a name that other objects can reference, and `command_line`, which defines the shell syntax of the command.

Command objects don't just refer to external programs; they capture the command syntax of external programs. The uppercase words surrounded by dollar signs are called macros. Macros are context-specific internal variables that Nagios replaces at runtime. The HOSTADDRESS macro, for example, refers to the host address of whatever host Nagios happens to be running this plug-in on.[8] This makes it possible to use one command definition for any host. Nagios will fork an exec of command_line exactly as it's written in the definition, but just before executing the command, Nagios replaces all the macros with their actual values.

To avoid unintended shell interpretation and injection attacks, Nagios strips certain characters out of the actual values before it replaces the macro keywords in the commands in some contexts.[9] For the same reasons, Nagios also prevents you from using special characters in host and service names. These characters are user-definable via directives in the nagios.cfg and you should be aware of them so you avoid using them in your definitions. As of the time of this writing, the illegal name characters are

```
` ~ ! $ % ^ & * | ' " < > ? , ( ) =
```

and the illegal macro output commands are

```
` ~ $ & | ' " < >
```

Different macros are available in different contexts, so it can be hard to know what macros you can use in every situation. For example, email-related macros, such as `CONTACTNAME`, are available to commands being run for notifications, but not commands being run for service checks. Check the online documentation at http://nagios.sourceforge.net/docs/3_0/macros.html for a complete matrix of available macros and the contexts they are available in.

In the nagios.cfg file is a directive called `resource_file`, which allows you to specify a file to create your own macros. This file is usually called resources.cfg or resource.cfg, and within it you may define up to 32 macros. The resources.cfg usually contains at least one macro, USER1, which resolves to the location of the installed plug-ins. Because resources.cfg is owned by root and read-only, it's a better place to define any usernames or passwords than the checkcommands.cfg, which is world readable. After they are set up in resources.cfg, your command objects can refer to the passwords by macro, thereby keeping them safe from prying eyes.

In Nagios 2.0 and later, macros are exported as environment variables, so any macro available to a command definition is also available to the program called by that definition. In Nagios 3.0 and later, you may also define "custom variables" inside host, service, and contact definitions. Custom variables may be defined by prefacing a directive with an underscore. If, for example, you wanted to track host MAC addresses in your host definitions, you could add the following directive to each host:

```
_macaddr            00:06:5B:A6:AD:AA;
```

To prevent name-space collisions, Nagios will convert your custom variable name to uppercase and prepend "HOST" "SERVICE" or "CONTACT" to it to create a unique macro. For example, the _macaddr custom variable we created below may be referenced as a macro using the name "$_HOSTMACADDR$". When the macro is exported to an environment variable, it will be further transformed into "$NAGIOS_HOSTMACADDR".

Contacts

Required for daemon start

Refers to: command, timeperiod, contactgroup

Referred to by: contactgroup

Listing 4.7 *Contact Example*

```
define contact{
   contact_name                    dave
   alias                           dave josephsen
   host_notification_period        24x7
   service_notification_period     work-hours
   host_notification_options       d,u,r,f
   service_notification_options    w,u,c,r,f
   host_notification_commands      host-email, send-sms
   service_notification_commands   service-email
   email                           dave@somewhere.org
   pager                           555-1024
   address1                        dave_josephsen@gmail.com
   address2                        cn=djosephs,ou=foo,dc=bar,dc=com
}
```

The contact object defines everything Nagios needs to know about a person. The `name` and `alias` directives provide the usual. Two timeperiod objects may be specified: the hours during which the contact wants to be notified of host problems and those during which the