SPK and Associates

www.spkaa.com
Ph: 888-310-4540

SPK and Associates
900 E Hamilton Ave, Ste.100
Campbell, CA 95008

Creating Custom Nagios Plugins

Nagios is an invaluable tool to help monitor a customer's infrastructure. Last year, I explained how easy it was to be able to integrate Network Appliance health checks. In fact, Nagios provides a plethora of out of the box plugins. Additional plugins are available via the Nagios Exchange site. The most common applications and protocols can be monitored without any additional software or custom coding. However, Nagios is just as easily extensible for any checks that you'd like to perform that aren't readily available.

As part of SPK's network management service, we find that all customers have very specific environments. Not only is it common to find custom applications, but customers may also require a varying level of visibility into those apps. For instance, the out of the box check_http script not only can verify connectivity to a web server, but it also can verify a specific string that should be returned. But what we if we want dive much deeper? Can we tell if the application throws any critical back-end errors during the login attempt? What if logins never fail, but performance slowly degrades? How can we be alerted to such situations?

These situations call for functionality beyond what the out-of-the box plugins can provide. Writing a custom plugin might sound like a daunting task, but if you follow my simple guidelines and troubleshooting steps, you'll be off and running in no time. Read further on how you can create your own custom Nagios plugins.

Basic Nagios Plugin Flow

The basic flow in a typical setup is as follows. When troubleshooting why a plugin doesn’t work as expected, it’s important to understand the interaction between the Nagios server, the remote host, and your custom plugin.

- Nagios reads the service definition
- If your service resides on a remote machine, you’re likely using NRPE
- Nagios executes check_nrpe and calls the desired check
- NRPE is spawned on the remote system, running as the nagios or NRPE unprivileged user.
- NRPE executes the check as defined in nrpe.cfg
- The check defined in nrpe.cfg points to your custom code / script
- Your custom code / script performs its duty, and exits with a return code. It also sends to standard output the result (OK, WARNING, CRITICAL), along with a short description of the result.
- NRPE takes the return code along with the short description and relays it back to the Nagios server

Basic Nagios Plugin Guidelines

Nagios plugins can be written in any language of your choice. Since Nagios simply calls the plugin and reads its return code, it leaves you free to create the plugin however you wish. However, there are certain considerations to keep in mind as you develop your plugin.

- When developing your plugin, what may work running as root or as the application owner may not work when Nagios runs NRPE as its own unprivileged user. In this case, sudo will need to be leveraged.
- Return codes. Your plugin MUST return one of the 4 following return codes:
    - 0 - OK
    - 1 - WARNING
    - 2 - CRITICAL
    - 3 - UNKNOWN
- Standard output: This is what will appear on the Nagios “scoreboard”. Typically you will want to be as informative as possible. Include some basic statistics into the status line where applicable. For instance, “OK: Application has no errors”, or “CRITICAL: Found 2 errors in the logs”.
- Your script MUST capture all output, both standard output and standard error. We do not want any child processes to produce any uncaptured output either. In the end, the status line that we send to Nagios should be that status line and nothing else.
- Script runtime: Be mindful of how long your plugin takes to execute. If it runs longer than 10 seconds, you may run into timeout issues.
- Script resources: Be mindful of much of an impact your script makes on the target system. You don’t want your script to be the cause of the poor performance that itself is reporting!
- Think out of the box when testing your plugin. For example, does your script have dependencies? How reliable are those dependencies? Could you have written the plugin without making use of that dependency?
- Always test your plugin before putting it into production. False alarms are the last thing you want, and when plugins “cry wolf”, their credibility is diminished.

Simple Plugin Example

Let’s follow the process flow I outlined earlier. First, we start with the service description on the Nagios server. In our example, I want to be alerted when an application’s log contains a critical error or keyword.

```
define service{
        use                             generic-service
        host_name                       host1, host2
        service_description             App Log Check
        is_volatile                     0
        check_period                    24x7
        max_check_attempts              3
        normal_check_interval           5
        retry_check_interval            1
        contact_groups                 app_owner_email
        notification_interval           60
        notification_period            workhours
        notification_options            w,u,c,r
        check_command                   check_nrpe!check_app_logs
        }
```

Nothing out of the ordinary here yet. This is a typical service check that utilizes NRPE. Next, we'll move on to the target system's nrpe.cfg:

```
command[check_app_logs]=sudo /usr/local/nagios/libexec/check_app_logs.sh
```

Here's where we define our new custom plugin. In our situation, our app directories are owned by the app owner, and our unprivileged NRPE user can't read them. So we've created a sudoers entry to allow this. Finally, we look at our actual sample script:

```
#!/bin/bash

#  Script to look for critical errors in the app log
#

ERROR="OutOfMem"
LOG="/opt/app/logfile.txt"


tail -2000 $LOG | egrep -i $ERROR > /dev/null 2>&1

if [ $? -eq 0 ]; then
   echo "CRITICAL:  $ERROR found in log"
   exit 2
else
   echo "OK:  Nothing critical found in log"
   exit 0
fi
```

This example is oversimplified, but it does touch on a few key pieces. Any output is suppressed (from calling 'tail' and piping to grep), and the exit status messages are concise but informative. Starting with a basic script gives you a good starting point to test from. As you add functionality and complexity to the script, you can monitor where certain test cases may fail, and where excessive strain might be put on the system.

Mike Solinap
Sr. Systems Integrator
SPK & Associates