Sauna Documentation

Release 0.0.18

Nicolas Le Manchet

Feb 20, 2018

Contents

Release v0.0.18. (*Installation*)

Sauna is a lightweight daemon designed to run health checks and send the results to a monitoring server.

Sauna comes batteries included, it is able run many system checks (load, memory, disk...) as well as monitor applications (redis, memcached, puppet...). It is easily *extensible* to include your own checks and can even run the thousands of existing *Nagios plugins*.

Painless monitoring of your servers is just a pip install away:

```
pip install sauna
```

Getting started with sauna:

CHAPTER 1

Installation

Prerequisites

Sauna is written in Python 3, prior to use it you must make sure you have a Python 3 interpreter on your system.

Pip

If you are familiar with the Python ecosystem, you won't be surprised that sauna can be installed with:

```
$ pip install sauna
```

That's it, you can call it a day!

Debian package

A Debian package for Jessie is built with each release, you can find them on GitHub.

Download and install the deb package:

```
$ wget https://github.com/NicolasLM/sauna/releases/download/<version>/sauna_<version>-1_all.deb
$ dkpg -i sauna_<version>-1_all.deb || apt-get install -f
```

The configuration file will be located at `/etc/sauna.yml` and sauna will be launched automatically on boot.

Source Code

Sauna is developed on GitHub, you can find the code at NicolasLM/sauna.

You can clone the public repository:

```
$ git clone https://github.com/NicolasLM/sauna.git
```

Once you have the sources, simply install it with:

```
$ python setup.py install
```

If you are interested in writing your own checks, head up to the *development*.

Docker image

A Docker image is available on the Docker Hub. It allows to run sauna in a container:

```
$ docker pull nicolaslm/sauna
```

If you want to share the configuration between the host and the container using a volume, be carreful about file permissions.

The configuration file might contains some sensible data like password and should not be readable for everyone. But inside the container sauna runs as user *sauna* (uid 4343) and need to read the configuration file. To do so, the easiest way is to create a user on the host with the same uid (4343) and chown the configuration file with this user. Then you can mount the configuration file inside the container and run sauna with

```
$ docker run -v /etc/sauna.yml:/app/sauna.yml:ro nicolaslm/sauna
```

CHAPTER 2

Command Line Interface

After a successful installation, the `sauna` command can be used to run and administer Sauna. An explanation of the available commands and flags are given with `sauna --help`:

```
$ sauna --help
Daemon that runs and reports health checks

Usage:
  sauna [--level=<lvl>] [--config=FILE] [<command> <args>...]
  sauna sample
  sauna (-h | --help)
  sauna --version

Options:
  -h --help      Show this screen.
  --version      Show version.
  --level=<lvl>  Log level [default: warn].
  --config=FILE  Config file [default: sauna.yml].

Available commands:
  list-active-checks            Display the checks that sauna will run.
  list-available-checks         Display the available checks.
  list-available-consumers      Display the available consumers.
  register                      Register a server to OVH Shinken monitoring
  status                        Show the result of active checks.
```

When no command is given, sauna runs in the foreground. It executes and sends the checks until interrupted. Sauna can also run as a *service* in the background.

CHAPTER 3

Configuration

Location

Sauna configuration is made of a single yaml file. By default it loads `sauna.yml` in the current directory. You can load another configuration file with the `--config` switch:

```
$ sauna --config /etc/sauna.yml
```

Note: This configuration file might end up containing secrets to access your monitoring server. It is a good idea not to make it world readable. Only the user running sauna needs to be able to read it.

Quickstart

Sometimes simply editing a configuration file feels easier than reading documentation. You can generate a default configuration file:

```
$ sauna sample
Created file ./sauna-sample.yml
```

You can adapt this default configuration to fit your needs, when you are ready rename it and launch sauna:

```
$ mv sauna-sample.yml sauna.yml
```

Content

The configuration yaml file contains three parts:

- Generic parameters
- Active consumers
- Active plugins

Generic parameters

All these parameters can be left out, in this case they take their default value.

periodicity How often, in seconds, will checks be run. The default value of 120 means that sauna will run all checks every two minutes. Individual checks that need to run more or less often can override their `periodicity` parameter.

hostname The name of the host that will be reported to monitoring servers. The default value is the fully qualified domain name of your host.

extra_plugins A list of directories where *additional plugins* can be found. Defaults to no extra directory, meaning it does not load plugins beyond the core ones.

include A path containing other configuration files to include. It can be used to separate each plugin in its own configuration file. File globs are expanded, example `/etc/sauna.d/*.yml`.

concurrency How many threads can process the checks at the same time. The default value of 1 means sauna will run checks one by one. Note that activating the concurrency system will, by default, only allow 1 check with the same name to run at the same time.

logging Sauna writes logs to the standard output by default. The `logging` parameter allows to pass a custom logging configuration to change the log format, write logs to files, send them to syslog and much more. Check the *logging syntax* for the details.

Example:

```
---
periodicity: 10
extra_plugins:
  - /opt/sauna_plugins
```

Active consumers

A list of the consumers you want to process your checks. It defines how sauna will interact with your monitoring server(s).

Example:

```
---
consumers:

  - type: NSCA
    server: receiver.shinken.tld
    port: 5667
    timeout: 10
```

Many consumers can be active at the same time and a consumer may be used more than once.

Active plugins

A list of plugins and associated checks.

Example:

```
---
plugins:

# Usage of disks
- type: Disk
  checks:
    - type: used_percent
      warn: 80%
```

```
      crit: 90%
    - type: used_inodes_percent
      warn: 80%
      crit: 90%
      periodicity: 300
```

A plugin may be defined many times in the list. This allows to run the same checks with different configurations parameters.

Plugin parameters

Some plugins accept additional configuration options, for example:

```
- type: Redis
  checks: ...
  config:
    host: localhost
    port: 6379
```

Unfortunately the parameters accepted by each plugins are not yet documented.

Check parameters

type The kind of check as defined by the plugin. All types available are listed by the command `sauna list-available-checks`.

warn The warning threshold for the check.

crit The critical threshold for the check.

name Optional, overrides the default generated name of the check which is in the form `plugin_type`. It becomes necessary to override the name when more than one checks of the same plugin and type are defined simultaneously.

periodicity Optional, overrides the global periodicity for this check. Used to run a check at a different frequency than the others.

Logging syntax

By default Sauna writes logs with the level `WARNING` or the level passed by the `--level` flag in the command line to the standard output.

To further customize how logs are processed, Sauna can also leverage Python dictConfig. This allows the user to modify every aspect of the logging system, for instance:

- Storing the logs in a file rotating every week
- Silencing some log message but not others
- Forwarding logs to syslog
- Modifying the format of the logs

To do that a dictionary configuration must be passed in the `logging` parameter of the configuration file. For example to remove the date from the record and write the message to stderr:

```
---
logging:
  version: 1
  formatters:
    simple:
      format: '%(message)s'
  handlers:
    console:
      class: logging.StreamHandler
      formatter: simple
      stream: ext://sys.stderr
  root:
    level: DEBUG
    handlers: [console]
```

Make sure to read the Python logging documentation to go further.

CHAPTER 4

Launching on boot

You will probably want to launch sauna as a service as opposed as attached to a shell. This page presents a few possibilities for doing that.

Note: If you installed sauna via the Debian package, everything is already taken care of and you can skip this part.

Creating a user

Sauna does not need to run as root, following the principle of least privilege, you should create a user dedicated to sauna:

```
adduser --system --quiet --group --no-create-home sauna
```

Systemd

If your distribution comes with the systemd init system, launching sauna on boot is simple. Create a `systemd unit file` at `/etc/systemd/system/sauna.service`:

```
[Unit]
Description=Sauna health check daemon
Wants=network-online.target
After=network-online.target

[Service]
Type=simple
ExecStart=/opt/sauna/bin/sauna --config /etc/sauna.yml
User=sauna

[Install]
WantedBy=multi-user.target
```

Indicate in `ExecStart` the location where you installed sauna and in `User` which user will run sauna.

Enable the unit, as root:

```
# systemctl daemon-reload
# systemctl enable sauna.service
Created symlink to /etc/systemd/system/sauna.service.
```

```
# systemctl start sauna.service
# systemctl status sauna.service
 sauna.service - Sauna health check daemon
   Loaded: loaded (/etc/systemd/system/sauna.service; enabled)
   Active: active (running) since Sat 2016-05-14 13:13:16 CEST; 1min 17s ago
 Main PID: 30613 (sauna)
   CGroup: /system.slice/sauna.service
           -30613 /opt/sauna/bin/python3.4 /opt/sauna/bin/sauna --config /etc/sauna.yml
```

Supervisor

Supervisor is a lightweight process control system used in addition of init systems like systemd or SysVinit.

Create a supervisor definition for sauna:

```
[program:sauna]
command=/opt/sauna/bin/sauna --config /etc/sauna.yml
user=sauna
```

Load the new configuration:

```
$ supervisorctl reread
$ supervisorctl update
```

CHAPTER 5

Plugins

Plugins contains checks that sauna can run to determine the health of the system. Sauna will periodically run the checks contained in active plugins and forward the results to active *consumers*.

Plugins can either be core ones shipped with sauna, or *your own plugins* for monitoring the specific parts of your system.

Todo

Find a way to automatically document core plugins. For now you can find the list of core plugins on GitHub.

CHAPTER 6

Consumers

Consumers provide a way for sauna to process the checks generated by *plugins*. The most common job for a consumer is to send the results to a monitoring server, but it can go beyond that.

Consumers are enabled in *sauna.yml*

Todo

Find a way to automatically document core consumers. For now you can find the list of consumers on GitHub.

CHAPTER 7

Cookbook

This page provides recipes to get the best out of sauna.

HAProxy health checks

When load balancing servers with HAProxy you might want to enable health checks. If one of your servers is running out of memory, overloaded or not behaving properly you can remove it from the pool of healthy servers.

To help you achieve that sauna has a special consumer that listens on a TCP port and returns the status of the server when getting an incoming connection. This consumer is the `TCPServer` consumer.

Enabling the TCP server

To enable the TCP server add it to the list of *active consumers*:

```
---
consumers:

  TCPServer:
    port: 5555
```

Let's launch sauna and try to connect to the port 5555:

```
$ nc localhost 5555
OK
```

As the system is healthy sauna answers `OK`. Let's try by switching a check to `CRITICAL`:

```
$ nc localhost 5555
CRITICAL
```

Configuring HAProxy

We will configure HAProxy to remove a server from the pool as soon as it is not in `OK` state. For that we will use tcp-check.

Assuming you have a load balancing frontend/backend already set up, activate checks:

```
backend webfarm
    mode http
    option tcp-check
    tcp-check connect port 5555
    tcp-check expect string OK
    server web01 10.0.0.1:80 check
    server web02 10.0.0.2:80 check
    server web03 10.0.0.3:80 check
```

- `option tcp-check` enables level 3 health checks
- `tcp-check connect port 5555` tells HAProxy to check the port 5555 of servers in the pool
- `tcp-check expect string OK` consider the server down if it does not answer `OK`

Reusing Nagios plugins

Nagios plugins are still very popular, their simple API can be considered the de facto standard for monitoring checks. Sauna can run Nagios plugins through its `Command` plugin.

Here we will run the famous `check_http` for monitoring Google. Add a `Command` plugin to *sauna.yml*:

```
---
plugins:

  - type: Command
    checks:
      - type: command
        name: check_google
        command: /usr/lib/nagios/plugins/check_http -H www.google.com
```

Run sauna:

```
$ sauna
ServiceCheck(name='check_google', status=0, output='HTTP OK: HTTP/1.1 302 Found')
```

Note: Nagios plugins may be convenient but they rely on forking a process for each check. Consider using some of the lighter sauna core plugins if this is an issue.

Passive host checks

When it is not possible to check if a host is alive by sending a ping (for instance when the host is in a private network), Nagios and Shinken can use passive host checks submitted via NSCA.

Passive host checks work like normal service checks, except that they don't carry a service name:

```
---
plugins:

  - type: Dummy
    checks:
      - type: dummy
        name: ""
        status: 0
        output: Host is up and running
```

Configure your monitoring server to consider your host down if no passive host check has been received for one minute:

```
define host {
    address                 192.168.20.3
    host_name               test
    use                     generic-host
    check_command           check_dummy!2
    active_checks_enabled   0
    passive_checks_enabled  1
    check_freshness         1
    freshness_threshold     60
}
```

CHAPTER 8

Frequently Asked Questions

What is the licence?

Sauna is released under the `BSD license`.

Where does that name come from?

With more than 10M repositories on GitHub, do you know how hard is it to find a cool name for your new project?

Development guide:

CHAPTER 9

Writing custom checks

Sauna ships with its own plugins for standard system monitoring, but sometimes you need more. This guide is a quick tutorial to start writing your own Python plugins to extend sauna's checks.

If writing Python is not an option, binaries written is any language can be run through the *Command plugin*.

For the sake of learning we will create the Uptime plugin. It will contain a simple check that will alert you when the uptime for your machine is under a threshold. This could be used to get a notification when a server rebooted unexpectedly.

Custom plugins directory

Your custom plugins must live somewhere on your file system. Let's say `/tmp/sauna_plugins`:

```
$ mkdir /tmp/sauna_plugins
```

This directory will contain Python modules, like our `uptime.py`:

```
$ cd /tmp/sauna_plugins
$ touch uptime.py
```

The Uptime class

All plugins have in common the same Python class `Plugin`, so the quite not working simplest implementation of a plugin is:

```
from sauna.plugins import Plugin

class Uptime(Plugin):
    pass
```

This implementation must be registered into sauna to be used to launch checks, as often in Python, this is done through a bit of decorator magic:

```
from sauna.plugins import Plugin, PluginRegister

my_plugin = PluginRegister('Uptime')

@my_plugin.plugin()
```

```
class Uptime(Plugin):
    pass
```

Here we are, a minimal class that is a sauna plugin. Now let's create our check.

The uptime check

A check is simply a method of the Plugin that is marked as a check, again through a decorator:

```
@my_plugin.plugin()
class Uptime(Plugin):

    @my_plugin.check()
    def uptime(self, check_config):
        return self.STATUS_OK, 'Uptime looks good'
```

So far we have an `Uptime` plugin, with an `uptime` check that always returns a positive status. Here is a bit of convention about checks: they must return a tuple containing the status (okay, warning, critical or unknown) and a human readable string explaining the result.

Arguably this check is not really useful, let's change that by actually fetching the uptime from `/proc/uptime`:

```
@my_plugin.check()
def uptime(self, check_config):
    with open('/proc/uptime') as f:
        uptime_seconds = float(f.read().split()[0])
    return (self._value_to_status_more(uptime_seconds, check_config),
            'Uptime is {}'.format(timedelta(seconds=uptime_seconds)))
```

The `check_config` passed to your check method contains the information needed to run the check and generate a status, it contains for instance the warning and critical thresholds. The value of uptime in seconds can be compared to the threshold with `_value_to_status_more`, which returns the correct status.

If during the execution of the check an exception is thrown, for instance if the `/proc` file system is not available, the check result will have the status `unknown`.

The final plugin

All these snippets together give the final plugin code:

```
from datetime import timedelta

from sauna.plugins import Plugin, PluginRegister

my_plugin = PluginRegister('Uptime')

@my_plugin.plugin()
class Uptime(Plugin):

    @my_plugin.check()
    def uptime(self, check_config):
        with open('/proc/uptime') as f:
            uptime_seconds = float(f.read().split()[0])
        return (self._value_to_status_more(uptime_seconds, check_config),
                'Uptime is {}'.format(timedelta(seconds=uptime_seconds)))
```

Configuring sauna to use Uptime

In the last step of this tutorial you need to tell sauna where to find your plugin, this is done through the `extra_plugins` configuration parameter. It is a list of directories where sauna will look for modules:

```
---
periodicity: 10
extra_plugins:
  - /tmp/sauna_plugins

consumers:

  Stdout:

plugins:

  - type: Uptime
    checks:
      - type: uptime
        warn: 300
        crit: 60
```

You can verify that sauna found your plugin by listing the available checks:

```
$ sauna list-available-checks

Load: load1, load15, load5
Uptime: uptime
[...]
```

Finally run sauna:

```
$ sauna

ServiceCheck(name='uptime_uptime', status=0, output='Uptime is 4 days, 1:24:19.790000')
```

CHAPTER 10

Contributing

This page contains the few guidelines and conventions used in the code base.

Pull requests

The development of sauna happens on GitHub, the main repository is https://github.com/NicolasLM/sauna. To contribute to sauna:

- Fork `NicolasLM/sauna`
- Clone your fork
- Create a feature branch `git checkout -b my_feature`
- Commit your changes
- Push your changes to your fork `git push origin my_feature`
- Create a GitHub pull request against `NicolasLM/sauna`'s master branch

Note: Avoid including multiple commits in your pull request, unless it adds value to a future reader. If you need to modify a commit, `git commit --amend` is your friend. Write a meaningful commit message, see How to write a commit message.

Python sources

The code base follows pep8 guidelines with lines wrapping at the 79th character. You can verify that the code follows the conventions with:

```
$ pep8 sauna tests
```

Running tests is an invaluable help when adding a new feature or when refactoring. Try to add the proper test cases in `tests/` together with your patch. The test suite can be run with nose:

```
$ nosetests
....................................
----------------------------------------------------
Ran 36 tests in 0.050s

OK
```

Compatibility

Sauna runs on all versions of Python 3 starting from 3.2. Tests are run on Travis to ensure that. Except from a few import statements, this is usually not an issue.

Documentation sources

Documentation is located in the `doc` directory of the repository. It is written in reStructuredText and built with Sphinx.

For `.rst` files, the line length is 99 chars as opposed of the 79 chars of python sources.

If you modify the docs, make sure it builds without errors:

```
$ cd doc/
$ make html
```

The generated HTML pages should land in `doc/_build/html`.

CHAPTER 11

Internals

This page provides the basic information needed to start hacking sauna. It presents how it works inside and how the project is designed.

Design choices

Easy to install

Installing software written in Python can be confusing for some users. Getting a Python interpreter is not an issue but installing the code and its dependencies is.

The Python ecosystem is great, so many high quality libraries are available for almost anything. While it is okay for a web application to require dozens of dependencies, it is not for a simple monitoring daemon that should run on any Unix box.

Most of the times checks are really basics, they involve reading files, contacting APIs... Often these things can be done in one line of code using a dedicated library, or 10 using the standard library. The latter has the advantage of simplifying the life of the end user.

Of course it does not mean that sauna has to do everything from scratch, sometimes it's fine to get a little help. For instance the psutil library is so convenient for retrieving system metrics that it would be foolish not to rely on it. On the other hand getting statistics from memcached is just a matter of opening a socket, sending a few bytes and reading the result. It probably does not justify adding an extra dependency that someone may have a hard time installing.

Batteries included, but removable

Sauna tries its best to provide a set of core plugins that are useful to system administrators. But for instance, a user not interested in monitoring system metrics should be able to opt-out of system plugins. This reduces the footprint of sauna and doesn't require to user to install external dependencies that he will never use.

Python 3 only

Not supporting Python 2 simplifies the code base.

Python 3 has been around for about 8 years, it is available on every distribution and starts to replace Python 2 as the default interpreter on some of them. Most of the libraries in the Python ecosystem are compatible with the version 3.

Efficient

Sauna tries to consume as little resources as possible. Your server probably has more important things to do than checking itself.

Very often monitoring tools rely on launching external processes to fetch metrics from other systems. How often have you seen a program firing a `/bin/df -h` and parsing the output to retrieve the disk usage?

This puts pressure on the system which has to fork processes, allocate memory and handle context switches, while most of the times its possible to use a dedicated API to retrieve the information, in this case the `/proc` file system.

Concurrency

Main thread

The main thread is responsible for setting up the application, launching the workers, handling signals and tearing down the application. It creates one producer thread and some consumer threads.

Producer

The producer is really simple, it is a loop that creates instances of plugins, run the checks and goes to sleep until it needs to loop again. Check results are appended to the consumers' queues.

To handle checks that don't run at the same interval, a simple scheduler tells which checks should be run each time the producer wakes up.

Consumers

Consumers exits in two flavors: with and without a queue. Queued consumers are synchronous, when they receive a check in their queue they use it straight away. The `NSCAConsumer`, for instance, gets a check and sends it to a monitoring server.

Asynchronous consumers do not have a queue, instead when they need to know the status of a check, they read it in a shared dictionary containing the last instance of all checks. A good example is the `TCPServerConsumer`, it waits until a client connects to read the statuses from the dictionary.

Each consumer runs on its own thread to prevent one consumer from blocking another.

Thread safety

Queues are instances of `queue.Queue` which handles the locking behind the scenes.

Asynchronous consumers must only access the `check_results` shared dictionary after acquiring a lock:

```
with check_results_lock:
    # do somethin with check_results
```