Vaurien Documentation

Release 0.1

Mozilla

November 06, 2012

CONTENTS

Vaurien, the Chaos TCP Proxy

Ever heard of the Chaos Monkey?

It's a project at Netflix to enhance the infrastructure tolerance. The Chaos Monkey will randomly shut down some servers or block some network connections, and the system is supposed to survive to these events. It's a way to verify the high availability and tolerance of the system.

Besides a redundant infrastructure, if you think about reliability at the level of your web applications there are many questions that often remain unanswered:

- What happens if the MYSQL server is restarted? Are your connectors able to survive this event and continue to work properly afterwards?
- Is your web application still working in degraded mode when Membase is down?
- Are you sending back the right 503s when postgresql times out ?

Of course you can – and should – try out all these scenarios on stage while your application is getting a realistic load.

But testing these scenarios while you are building your code is also a good practice, and having automated functional tests for this is preferable.

That's where **Vaurien** is useful.

Vaurien is basically a Chaos Monkey for your TCP connections. Vaurien acts as a proxy between your application and any backend.

You can use it in your functional tests or even on a real deployment through the command-line.

CHAPTER

ONE

INSTALLING VAURIEN

You can install Vaurien directly from PyPI; the best way to do so is via *pip*:

```
$ pip install vaurien
```

CHAPTER

TWO

USING VAURIEN FROM THE COMMAND-LINE

Vaurien is a command-line tool.

Let's say you want to add a delay for 20% of the requests done on google.com:

```
$ vaurien --local localhost:8000 --distant google.com:80 --behavior 20:delay
```

Vaurien will stream all the traffic to google.com but will add delays 20% of the time. You can pass options to the handler using *–handlers.NAME.OPTION* options:

```
$ vaurien --local localhost:8000 --distant google.com:80 --behavior 20:delay \
    --handlers.delay.sleep 2
```

Passing all options through the command-line can be tedious, so you can also create a *ini* file for this:

```
[vaurien]
distant = google.com:80
local = localhost:8000
behavior = 20:delay

[handler:delay]
sleep = 2
```

Each behavior applied on the request or response going through Vaurien is called a **handler**, and the ini file gets one section per handler.

You can find a descriptions of all built-in handlers here: *Handlers*.

CHAPTER

THREE

CONTROLLING VAURIEN LIVE

Sometimes, it is useful to control live the proxy, so you can change its behavior live between client calls.

Vaurien provides an HTTP server with a few APIs, which can be used to control the proxy.

To activate it, use the *–http* option:

```
$ vaurien --http
```

By default the server runs on port **8080** while the proxy runs on **8000**

Once it runs, you can call it using cURL or any HTTP client. See the *APIs*.

CHAPTER

FOUR

CONTROLLING VAURIEN IN YOUR CODE

If you want to run and drive a Vaurien proxy from your code, the project provides a few helpers for this.

For example, if you want to write a test that uses a Vaurien proxy, you can write:

```
import unittest
from vaurien import Client, start_proxy, stop_proxy


class MyTest(unittest.TestCase):

    def setUp(self):
        self.proxy_pid = start_proxy(port=8080)

    def tearDown(self):
        stop_proxy(self.proxy_pid)

    def test_one(self):
        client = Client()
        options = {'inject': True}

        with client.with_handler('error', **options):
            # do something...
            pass

        # we're back to normal here
```

In this test, the proxy is started and stopped before and after the test, and the Client class will let you drive its behavior.

Within the **with** block, the proxy will error out any call by using the *errors* handler, so you can verify that your application is behaving as expected when it happens.

CHAPTER

FIVE

EXTENDING VAURIEN

Vaurien comes with a handful of useful *Handlers*, but you can create your own handlers and plug them in a configuration file.

In fact that's the best way to create realistic issues. Imagine that you have a very specific type of error on your LDAP server everytime your infrastructure is under heavy load. You can reproduce this issue in your handler and make sure your web application behaves as it should.

Creating new handlers is done by implementing a class with a specific signature.

You can inherit from the base class Vaurien provides and just implement the **__call__** method:

```
from vaurien.handlers import BaseHandler

class MySuperHandler(BaseHandler):

    name = 'super'
    options = {}

    def __call__(self, client_sock, backend_sock, to_backend):
        # do something here
```

More about this in *Writing Handlers*.

CHAPTER

SIX

CODE REPOSITORY

If you're interested to look at the code, it's there: https://github.com/mozilla-services/vaurien

Don't hesitate to send us pull requests or to open issues!

CHAPTER

SEVEN

MORE DOCUMENTATION

Contents:

7.1 APIs

GET /handler

Returns the current handler in use.

Example:

```
$ curl http://localhost:8080/handler
normal
```

POST /handler

Set the handler.

Example:

```
$ curl -d"delay" http://localhost:8080/handler
OK
```

GET /handlers

Returns a list of handlers that are possible to use

Example:

```
$ curl http://localhost:8080/handlers
{"handlers": ["delay", "error", "hang", "blackout", "dummy"]}
```

7.2 Command line

You can use these APIs directly from the command-line using the *vaurienctl* cli tool.

With it, you can either list the available handlers, get the current one or set the handler to another one. Here is a quick demo:

```
$ vaurienctl list-handlers
delay, error, hang, blackout, dummy
$ vaurienctl set-handler blackout
Handler changed to "blackout"
```

```
$ vaurienctl get-handler
blackout
```

7.3 Handlers

Vaurien provides a collections of handlers.

7.3.1 blackout

Just closes the client socket on every call.

7.3.2 delay

Adds a delay before the backend is called.

The delay can happen *after* or *before* the backend is called.

Options:

- **before**: If True adds before the backend is called. Otherwise after (bool, default: True)
- **sleep**: Delay in seconds (int, default: 1)

7.3.3 dummy

Dummy handler.

Every incoming data is passed to the backend with no alteration, and vice-versa.

7.3.4 error

Reads the packets that have been sent then send random data in the socket.

The *inject* option can be used to inject data within valid data received from the backend. The Warmup option can be used to deactivate the random data injection for a number of calls. This is useful if you need the communication to settle in some speficic protocols before the ramdom data is injected.

Options:

- **inject**: Inject errors inside valid data (bool, default: False)
- **warmup**: Number of calls before erroring out (int, default: 0)

7.3.5 hang

Reads the packets that have been sent then hangs.

Acts like a *pdb.set_trace()* you'd forgot in your code ;)

7.4 Writing Handlers

Creating new handlers is done by implementing a class with a specific signature.

You can inherit from the base class Vaurien provides and just implement the **__call__** method:

```
from vaurien.handlers import BaseHandler

class MySuperHandler(BaseHandler):

    name = 'super'
    options = {}

    def __call__(self, client_sock, backend_sock, to_backend):
        # do something here
```

Vaurien can use this handler and call it everytime data is being seen on one hand or the other.

Where:

- **name** - the name under which your backend is known
- **options** - a mapping containing your handler options
- **client_sock** - the socket opened with the client
- **backend_sock** - the socket opened with the backend server
- **to_backend** - a boolean giving the direction of the call. If True it means some data is available in the client socket, that is supposed to go to the backend. If False, it means data is available on the backend socket and should be tramsmitted back to the client.

A handler instance is initialized with two values:

- **settings** - the settings loaded for the handler
- **proxy** - the proxy instance

For the handler options, each option is defined in the **options** mapping. The key is the option name and the value is a 3-tuple providing:

- a description
- a type
- a default value

every option is optional and need a default value

7.4.1 Full handler example

Here is how the *delay* handler is specified:

```
from vaurien.handlers import BaseHandler

class Delay(BaseHandler):
    """Adds a delay before the backend is called.
    """
    name = 'delay'
    options = {'sleep': ("Delay in seconds", int, 1),
               'before':
                       ("If True adds before the backend is called. Otherwise"
```

```
                    " after", bool, True)}

    def __call__(self, client_sock, backend_sock, to_backend):
        before = to_backend and self.options('before')
        after = not to_backend and not self.options('before')

        if before:
            gevent.sleep(self.options('sleep'))

        data = self._get_data(client_sock, backend_sock, to_backend)

        if after:
            gevent.sleep(self.options('sleep'))

        if data:
            dest = to_backend and backend_sock or client_sock
            dest.sendall(data)
```

7.4.2 Using handlers

Once the handler is ready, you can point it to Vaurien by providing its fully qualified name - e.g. the class name prefixed by the module and package(s) names.

Then you can use it with the **–behavior** option:

```
$ vaurien --local localhost:8000 --distant google.com:80 \
    --behavior 20:path.to.the.callable \
    --handlers.delay.sleep 2
```

Or by using a configuration file:

```
[vaurien]
behavior = 20:foobar

[handler:foobar]
callable = path.to.the.callable
foo=bar
```

And calling Vaurien with –config:

```
$ vaurien --config config.ini
```