Vaurien Documentation

Release 0.4

Mozilla

November 16, 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

DESIGN

Vaurien is a TCP proxy that simply reads data sent to it and pass it to a backend, and vice-versa.

It has built-in **protocols**: Tcp, Http, Redis & Memcache. The **Tcp** protocol is the default one and just sucks data on both sides and pass it along.

Having higher-level protocols is mandatory in some cases, when Vaurien needs to read a specific amount of data in the sockets, or when you need to be aware of the kind of response you're waiting for, and so on.

Vaurien also has **behaviors**. A behavior is a class that's going to be invoked everytime Vaurien proxies a request. That's how you can impact the behavior of the proxy. For instance, adding a delay or degrading the response can be implemented in a behavior.

Both **protocols** and **behaviors** are plugins, allowing you to extend Vaurien by adding new ones.

Last, but not least, Vaurien provides a couple of APIs you can use to change the behavior of the proxy live. That's handy when you are doing functional tests against your server: you can for instance start to add big delays and see how your web application reacts.

CHAPTER

THREE

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 HTTP requests made on **google.com**:

```
$ vaurien --protocol http --proxy localhost:8000 --backend google.com:80 \
        --behavior 20:delay
```

With this set up, Vaurien will stream all the traffic to **google.com** by using the *http* protocol, and will add delays 20% of the time.

You can find a description of all built-in protocols here: *Protocols*.

You can pass options to the behavior using *–behavior-NAME-OPTION* options:

```
$ vaurien --protocol http --proxy localhost:8000 --backend google.com:80 \
    --behavior 20:delay \
    --behavior-delay-sleep 2
```

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

```
[vaurien]
backend = google.com:80
proxy = localhost:8000
protocol = http
behavior = 20:delay

[behavior:delay]
sleep = 2
```

You can find a description of all built-in behaviors here: *Behaviors*.

You can also find some usage examples here: *Examples*.

CHAPTER

FOUR

CONTROLLING VAURIEN LIVE

Vaurien provides an HTTP server with a few APIs, which can be used to control the proxy and change its behavior on the fly.

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

```
$ vaurien --http
```

By default the server runs on **locahost:8080** but you can change it with the **–http-host** and **–http-port** options.

See *APIs* for a full list of APIs.

CHAPTER

FIVE

CONTROLLING VAURIEN FROM 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_behavior('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* behavior, so you can verify that your application is behaving as expected when it happens.

CHAPTER

SIX

EXTENDING VAURIEN

Vaurien comes with a handful of useful *Behaviors* and *Protocols*, but you can create your own ones 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 behavior and make sure your web application behaves as it should.

Creating new behaviors and protocols is done by implementing classes with specific signatures.

For example if you want to create a **super** behavior, you just have to write a class with two special methods: **on_before_handle** and **on_after_handle**.

Once the class is ready, you can register it with **Behavior.register**:

```
from vaurien.behaviors import Behavior

class MySuperBehavior(object):

    name = 'super'
    options = {}

    def on_before_handle(self, protocol, source, dest, to_backend):
        # do something here
        return True

     def on_after_handle(self, protocol, source, dest, to_backend):
        # do something else
        return True

Behavior.register(MySuperBehavior)
```

You will find a full tutorial in *Extending Vaurien*.

CHAPTER

SEVEN

CONTRIBUTE

The code repository & bug tracker are located at https://github.com/mozilla-services/vaurien

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

CHAPTER

EIGHT

MORE DOCUMENTATION

Contents:

8.1 APIs

GET /behavior

Returns the current behavior in use.

Example:

```
$ curl -XGET http://localhost:8080/behavior
{
  "behavior": "dummy"
}
```

POST /behavior

Set the behavior. The behavior must be provided in a JSON mapping in the body of the request, with a **name** key for the behavior name, and any option to pass to the behavior class.

Example:

```
$ curl -d '{"sleep": 2, "name": "delay"}' http://localhost:8080/behavior \
      -H "Content-Type: application/json"
 {
   "status": "ok"
 }
```

GET /behaviors

Returns a list of behaviors that are possible to use

Example:

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

8.2 Command line

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

vaurienctl can be used to list the available behaviors, get the current one, or set it.

Here is a quick demo:

```
$ vaurienctl list-behaviors
delay, error, hang, blackout, dummy

$ vaurienctl set-behavior blackout
Behavior changed to "blackout"

$ vaurienctl get-behavior
blackout
```

8.3 Behaviors

Vaurien provides a collections of behaviors.

8.3.1 blackout

Reads the packets that have been sent then hangs.

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

8.3.2 delay

Adds a delay before or after 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)

8.3.3 dummy

Transparent behavior. Nothing's done.

8.3.4 error

Reads the packets that have been sent then send back "errors".

Used in cunjunction with the HTTP Procotol, it will randomly send back a 501, 502 or 503.

For other protocols, it returns random data.

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.

The *inject* option is deactivated when the *http* protocol is used.

Options:

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

8.3.5 hang

Reads the packets that have been sent then hangs.

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

8.4 Protocols

Vaurien provides a collections of protocols.

8.4.1 http

HTTP protocol.

Options:

- **buffer**: Buffer size (int, default: 2048)
- **keep_alive**: Keep the connection alive (bool, default: False)
- **reuse_socket**: If True, the socket is reused. (bool, default: False)

8.4.2 memcache

Memcache protocol.

Options:

- **buffer**: Buffer size (int, default: 2048)
- **keep_alive**: Keep the connection alive (bool, default: False)
- **reuse_socket**: If True, the socket is reused. (bool, default: False)

8.4.3 redis

Redis protocol.

Options:

- **buffer**: Buffer size (int, default: 2048)
- **keep_alive**: Keep the connection alive (bool, default: False)
- **reuse_socket**: If True, the socket is reused. (bool, default: False)

8.4.4 tcp

TCP handler.

Options:

- **buffer**: Buffer size (int, default: 2048)
- **keep_alive**: Keep the connection alive (bool, default: False)
- **reuse_socket**: If True, the socket is reused. (bool, default: False)

8.5 Extending Vaurien

Note: Before reading this section, make sure you read *Keep-alive vs Disconnect*

You can extend Vaurien by writing new **protocols** or new **behaviors**.

8.5.1 Writing Protocols

XXX

8.5.2 Writing Handlers

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

```
from vaurien.handlers import Handler

class MySuperHandler(object):

    name = 'super'
    options = {}

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


Handler.register(MySuperHandler)
```

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

You must call **Handler.register** against your class is order to add it to the list of the available plugins.

Let's see the different attributes and options we have in this class:

- **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.

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

Everytime a handler is used, it gets two extra attributes:

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

8.5.3 The BaseHandler class

XXX

8.5.4 Full handler example

Here is how the *delay* handler is specified:

```
from vaurien.handlers.base import BaseHandler


class Dummy(BaseHandler):
    """Dummy handler.

    Every incoming data is passed to the backend with no alteration,
    and vice-versa.
    """
    name = 'dummy'
    options = {'keep_alive': ("Keep-alive protocol",
                              bool, False),
        'reuse_socket': ("If True, the socket is reused.",
                              bool, False)}

    def __call__(self, client_sock, backend_sock, to_backend):
        data = self._get_data(client_sock, backend_sock, to_backend)
        if data:
            dest = to_backend and backend_sock or client_sock
            source = to_backend and client_sock or backend_sock
            dest.sendall(data)

            # If we are not keeping the connection alive
            # we can suck the answer back and close the socket
            if not self.option('keep_alive'):
                data = ''
                while True:
                    data = dest.recv(1024)

                    if data == '':
                        break
                    source.sendall(data)
                dest.close()
                dest._closed = True
        elif not to_backend:
            # We want to close the socket if the backend sock is empty
            if not self.option('reuse_socket'):
                backend_sock.close()
                backend_sock._closed = True

        return data != ''
```

8.5.5 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 --proxy localhost:8000 --backend google.com:80 \
    --behavior 20:path.to.the.callable \
    --handler-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
```

8.6 Keep-alive vs Disconnect

Explain here the different strategies

8.7 Examples

Proxying on an HTTP backend and sending back 50x errors 20% of the time:

```
$ vaurien --proxy 0.0.0.0:8888 --backend blog.ziade.org:80 --behavior 20:error --handler-error-proto
```

Adding a 1 second delay on every call to a MySQL server:

```
$ vaurien --proxy 0.0.0.0:3307 --backend 0.0.0.0:3306 --stay-connected --behavior 100:delay \
    --handler-delay-sleep 1
```