Fuzzing Documentation

Release 0.3.2

Stefan Braun

Jun 10, 2018

Contents

Contents:

CHAPTER 1

Release Notes

Release 0.3.1

No functional changes. Only tested and released for Python 3.5.

Release 0.3.0

New features:

- Run multiple tests in parallel on multiple processors. Number of processors and processes is configurable.
- Test statistics of the processes are merged and printed.

API changes:

- `FuzzExecutor.stats` returns an instance of `TestStatCounter`, not a simple dict anymore.

You may want to look into `TestStatCounter` and `Status`. See also `run_fuzzer.py` for intended usage.

Release 0.2.3

- Data structure for run statistics improved.
- Tests can now be configured using a YAML file.
- **Test runner script added for improved user experience ::** run_fuzzer.py config.yaml

Reading the test runner script may help to get a clearer picture how to use the tool.

Release 0.2.3a1

Package structure simplified.

Release 0.2.2

Mainly cleanup.

- Test uses pure Python test app. See `features/resources/testfuzz.py`.

Release 0.2.1

- Class `LoggerFactory`. Logger factory for configuration of the Python logging framework.
- The `fuzzer` module uses logging.

- Singleton decorator behaves much nicer since using `wrapt`. See Graham Dumpleton's talk on the workings of wrapt.

Release 0.2.0

Improved fuzz testing.

- Class `FuzzExecutor` makes fuzz testing of applications taking data files easy.

Release 0.1.0

First small step.

- Basic functions for fuzz testing.
- Decorator to declare a class as Singleton.

CHAPTER 2

Tutorial

The following sections will show how to use the classes and functions of the package.

2.1 Random testing

Systematic testing helps us to cover classes of equivalent test cases. Specifying those test classes largely reduces the effort for testing without sacrificing test coverage.

One drawback of this approach is that we're testing only what we expect to break. This may allow defects caused by unexpected side effects or unexpected input data to pass the tests . . . and show up in production systems.

Random testing is an approach to increase the coverage of the domain of our software's inputs by automatically running large amounts of tests with randomized input data. This might be totally random 'byte noise', mostly valid data provided by a carefully crafted generator, or anything in between.

Charlie Miller did some interesting work on fuzz testing. The function `fuzzer()` is essentially taken from *Babysitting an Army of Monkeys* (see references below).

References:

- http://fuzzinginfo.files.wordpress.com/2012/05/cmiller-csw-2010.pdf
- https://cansecwest.com/csw08/csw08-miller.pdf

2.1.1 How to do random testing on your own?

Fuzz testing can be done on different levels:

- unit (e.g. function, class, module),

- integration (components built from units),
- system (e.g. application).

In each case you need to provide a source for test data, call your SUT, and check the result. Put this into a loop and start fuzzing.

This is already good for robustness tests. In most cases you also want a kind of statistics and a documentation of the test cases resulting in an error.

2.1.2 Generating test data

In general random testing can be done with any kind of input data (I guess ;-). The code found in `fuzzing.fuzzer.fuzzer()` is working on a binary buffer. It is a copy of Charlie Miller's code mentioned above.

The binary buffer may contain something like a pdf, an image, a presentation and so on. It also works fine for normal text, covering ASCII texts, HTML, XML, JSON and other text based formats. `fuzzing.fuzzer.fuzz_string()` is a wrapper simplifying such use cases a bit.

2.1.3 Example of a simple generator:

Example

```
import fuzzing
seed = "This could be the content of a huge text file."
number_of_fuzzed_variants_to_generate = 10
fuzz_factor = 7
fuzzed_data = fuzzing.fuzz_string(seed, number_of_fuzzed_variants_to_generate, fuzz_
↪factor)
print(fuzzed_data)
```

Of course you can also create one fuzzed variant at a time and feed it directly into the SUT.

2.1.4 Calling the SUT with the test data

How to call the SUT depends obviously on its type. A Python function can be called directly with the created data. It might make sense to enclose the call into a try / except block to catch errors. It is also easy to check the result value for failure.

Testing software written in other languages works in the same way. You may want to write the fuzz generator in the target language, or just create the test data with Python and put it into a file for use by the target system.

Applications reading files can be tested creating fuzzed files in the same manner as described above: Read a valid seed file into a buffer, fuzz it and write it back to a new file. Then run the application in a separate process for each fuzzed file. In this case it is not that easy to gather useful information about the success or failure of the run. At least crashes are easily recognized.

2.1.5 The oracle - or: How to evaluate the test result?

The function evaluating the result of a test run is called *oracle*. That's fine because the result is not always clear and understandable ;-).

Running an application in a separate process as described above let us quite easily detect crashes. If we need more detailed information there is no general way to get at it. One of the most general information is a crash dump of the SUT.

Detecting issues not leading to a crash depends largely on the application we are looking at. If it creates some accessible output, like a processed file or a log file, we may be able to write parsers that enable us to look for failures.

2.1.6 Complete example:

The following sample code runs 100 tests against the applications listed in `apps_under_test`. Test data is generated using a simple fuzzer on a set of files defines in `file_list`.

After finishing the test runs a statistic is printed.

Note that `num_tests` should be much bigger for real testing. But it makes sense to start with a small number to get the test harness working. Then increase this number to a couple of millions or so.

Some of the code found in the `fuzzer` module is inlined for easier comprehension.

```
import math
import random
import subprocess
import time
import os.path
from tempfile import mkstemp
from collections import Counter


# Files to use as initial input seed.
file_list = ["./data/pycse.pdf", "./data/PyOPC.pdf", "./data/003_overview.pdf",
             "./data/Clean-Code-V2.2.pdf", "./data/GraphDatabases.pdf",
             "./data/Intro_to_Linear_Algebra.pdf", "./data/zipser-1988.pdf",
             "./data/QR-denkenswert.JPG"]

# List of applications to test.
apps_under_test = ["/Applications/Adobe Reader 9/Adobe Reader.app/Contents/MacOS/
↪AdobeReader",
                   "/Applications/PDFpen 6.app/Contents/MacOS/PDFpen 6",
                   "/Applications/Preview.app/Contents/MacOS/Preview",
                   ]


fuzz_factor = 50  # 250
num_tests = 100

# ##### End of configuration #####

def fuzzer():
    """Fuzzing apps."""
    stat_counter = Counter()
    for cnt in range(num_tests):
        file_choice = random.choice(file_list)
        app = random.choice(apps_under_test)
        app_name = app.split('/')[-1]
        file_name = file_choice.split('/')[-1]

        buf = bytearray(open(os.path.abspath(file_choice), 'rb').read())

        # Charlie Miller's fuzzer code:
        num_writes = random.randrange(math.ceil((float(len(buf)) / fuzz_factor))) + 1

        for _ in range(num_writes):
```

(continues on next page)

(continued from previous page)

```
            r_byte = random.randrange(256)
            rn = random.randrange(len(buf))
            buf[rn] = r_byte
        # end of Charlie Miller's code

        fd, fuzz_output = mkstemp()
        open(fuzz_output, 'wb').write(buf)

        process = subprocess.Popen([app, fuzz_output])

        time.sleep(1)
        crashed = process.poll()
        if crashed:
            logger.error("Process crashed ({} <- {})".format(app, file_choice))
            stat_counter[(app_name, 'failed')] += 1
        else:
            process.terminate()
            stat_counter[(app_name, 'succeeded')] += 1
    return stat_counter

if __name__ == '__main__':
    stats = fuzzer()
    print(stats)
```

2.1.7 Using FuzzExecutor

Fuzz testing applications using files can be used often because it is quite generic. Therefore it makes sense to encapsulate this functionality and make it easy to apply.

The example above can be written much faster using the class `FuzzExecutor`:

```
from fuzzing.fuzzer import FuzzExecutor

# Files to use as initial input seed.
file_list = ["./features/data/t1.pdf", "./features/data/t3.pdf", "./features/data/t2.
↪jpg"]

# List of applications to test.
apps_under_test = ["/Applications/Adobe Reader 9/Adobe Reader.app/Contents/MacOS/
↪AdobeReader",
                   "/Applications/PDFpen 6.app/Contents/MacOS/PDFpen 6",
                   "/Applications/Preview.app/Contents/MacOS/Preview",
                   ]

number_of_runs = 13

def test():
    fuzz_executor = FuzzExecutor(apps_under_test, file_list)
    fuzz_executor.run_test(number_of_runs)
    return fuzz_executor.stats

def main():
    stats = test()
    print(stats)
```

2.1.8 Getting test statistics

The property `FuzzExecutor.stat` is an instance of `TestStatCounter`. It provides the number of successful and failed runs for each application.

To combine the statistics of multiple test runs `TestStatCounter` implements __add__:

```
// Run multiple tests yielding a set stats = set(c1, c2, c3) of stat counters
...
// Then merge these counters to get a complete statistics of your test runs.
// This operation does not modify c1 to c3.
combined_stats = TestStatCounter(set())
for stat in stats:
    combined_stats += stat
```

`Status` is an enum class providing the supported values for test status:

```
@enum.unique
class Status(enum.Enum):
    """Status values for test runs."""
    FAILED = 0
    SUCCESS = 1
```

2.1.9 Running tests without coding

When running different sets of tests writing a script for each configuration is tedious. It would be nice to just write a configuration and feed it to a generic test runner.

`run_fuzzer.py` now reads a test configuration written using YAML notation. Please note that each process will execute `runs` tests. Therefore the number of executed tests is the product of `runs` and `processes`.

```
version: 1
seed_files: ['requirements.txt', 'README.rst']
applications: ['python & features/resources/testfuzz.py -p 0.3',
               '/Applications/Adobe Reader 9/Adobe Reader.app/Contents/MacOS/
↪AdobeReader']
runs: 4
processors: 3
processes: 8
```

If you want to run a couple of tests, just provide those configuration files and execute `run_fuzzer.py`. For example:

```
$ run_fuzzer.py test_config_one_processor.yaml
$ run_fuzzer.py test_config_4_processors.yaml
```

Each call to `run_fuzzer.py` will execute the tests as configured. It creates a `ProcessPoolExecutor` with pool size defined by the number of specified processors. The number of processes is (kind of) independent of the number processors; if there are more processes than processors, a new process will be started as soon as a processor is available.

If for example 2 processors and 5 processes are specified, not more than 2 processes will run in parallel at each point in time.

After executing all tests `run_fuzzer.py` merges the results of all processes and prints statistics like that:

```
__________________________________________________
Test Results:
__________________________________________________
Tests run/succeeded/failed: 32 / 25 / 7
AdobeReader
    FAILED: 0
    SUCCESS: 14
python
    FAILED: 7
    SUCCESS: 11

__________________________________________________
```

2.2 Logging

The Python standard library provides good logging capabilities with the module `logging`. Requirements on logging depend on the application and may change during the life cycle. Therefore a logging system must be flexible. The `logging` module is highly configurable and extendable.

Class `fuzzing.LoggerFactory` reads a YAML configuration file and initializes the logging system. It is just a thin layer on top of `logging` abstracting from the details of initialization. Loggers can be used as usual; the use of `LoggerFactory` is transparent for the loggers.

2.2.1 Configuration file

`LoggerFactory` expects to get a YAML file containing the configuration of the loggers. To deploy your configuration put it into a folder of your package, e.g.:

```
<my_package>
    <my_sources>
    <resources>
        log_config.yaml
    <tests>
    <doc>
```

Then add it to your MANIFEST.in so that it will be packaged with your code:

```
include <my_package>/resources
```

The documentation of the Python standard library describes how to write such a file: https://docs.python.org/3.4/library/logging.config.html.

Example:

```
version: 1
formatters:
  concise:
    format: '%(asctime)s - %(levelname)s - %(message)s'
  detailed:
    format: '%(asctime)s - %(levelname)s - %(filename)s:%(lineno)3d - %(funcName)s()␣
↪:: %(message)s'
  thread_info:
    format: '%(asctime)s - %(levelname)s - %(filename)s:%(lineno)3d - %(funcName)s() -
↪ %(thread)d:%(threadName)s :: %(message)s'
```

(continues on next page)

(continued from previous page)

```
handlers:
  console:
    class: logging.StreamHandler
    level: WARNING
    formatter: concise
    stream: ext://sys.stdout
  file:
    class: logging.handlers.RotatingFileHandler
    filename: 'fuzzer.log'
    maxBytes: 100000
    backupCount: 3
    level: DEBUG
    formatter: detailed
loggers:
  fuzzing:
    level: DEBUG
    handlers: [console, file]
    propagate: no
  fuzzing.fuzzing:
    level: INFO
    handlers: [console, file]
    propagate: no
  fuzzing.fuzzing.FuzzExecutor:
    level: INFO
    handlers: [file, console]
    propagate: no
root:
  level: WARNING
  handlers: [console]
```

2.2.2 Initialization

The `logging` system must be initialized before the first use. So put something like the following into the startup code of your application:

```
from gp_tools import LoggerFactory

def my_main():
    lf = LoggerFactory(package_name='my_package', config_file='resources/log_config.
↪yaml')
    lf.initialize()
```

Now you can log as you're used to it:

```
import logging

# 'my_logger' is the name as used in the configuration.
logger = logging.getLogger('my_logger')

logger.info('Happy Logging!')
```

That's it :-)

2.3 Singletons

Singleton classes are characterized by the fact that there will be never more than a single instance. This may be useful for classes handling physical devices or any other stateful objects, e.g. caches, that need to be handled in a consistent way.

Singletons should be used with care, because they may lead to high coupling if used in many places. So they may be comfortable first, but become a nightmare later on when extending or maintaining an application.

Creating a singleton class using the singleton decorator is simple:

```
from gp_decorators.singleton import singleton

@singleton
class SomeClass(object):
    """A singleton class."""
    # <your code>
```

CHAPTER 3

API Reference

3.1 Fuzzing

CHAPTER 4

License

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index