Introduction to Fuzzing in Python with AFL

Mon, Apr 13, 2015

Fuzzing is a technique in computer testing and security where you generate a bunch of random inputs, and see how some program handles it. For example, if you had a JPEG parser, you might create a bunch of valid images and broken images, and make sure it either parses them or errors out cleanly. In C (and other memory unsafe languages) fuzzing can often be used to discover segfaults, invalid reads, and other potential security issues. Fuzzing is also useful in Python, where it can discover uncaught exceptions, and other API contract violations.

This blog post is going to walk you through getting started with afl (American Fuzzy Lop), a new, but extremely powerful fuzzer which can be used on Python code. afl is very good at finding bugs. In addition to clever techniques for generating random inputs, it also instruments your program, and uses coverage data about what paths are being taken to find interesting new bugs.

To get started, you'll first need to install afl. If you're on OS X and using Homebrew:

```
$ brew install afl-fuzz
```

Next you'll probably want to create a virtualenv. Inside of it you'll want to grab a copy of the Python afl tooling and install it into the virtualenv:

```
$ pip install python-afl
```

Now you'll want to write a small script which takes some input from `sys.stdin`, and tries to parse it (or do whatever your program does):

```
import sys

import afl

from cryptography.hazmat.primitives.asymmetric.utils import (
    decode_rfc6979_signature
)

afl.init()

try:
    decode_rfc6979_signature(sys.stdin.read())
except ValueError:
    pass
```

Pretty simple. The `afl.init()` line is a thing for performance, you should put that line after all your imports and setup, before you do anything with `sys.stdin`. You need to make sure that you catch any exception which your code could throw, in `decode_rfc6979_signature` the only exception that it's documented as raising is `ValueError`, so anything else that's raised is an error. Please note that this will only be effective if your code is pure-python, if the majority of your code is a C extension (e.g. `cPickle`) you need to take a slightly different approach, which isn't described here.

Next you'll need to create a corpus of "example" inputs. It doesn't have to be big, just a few small examples of what a valid input to your function looks like. I've used as little as one input with success. Once you've got them, put them all in a directory, one per file (the files' names don't matter).

Now you can invoke afl:

```
$ py-afl-fuzz -o results-path/ -i /path/to/examples -- /path/to/python /path/to/your/test/sc
ript.py
```

It'll pop up a curses display to let you know the progress, and you're off to the traces. Some of the display is self-explanatory, for the rest you can consult the documentation. When afl discovers crashers, you'll be able to find them in `results-path/crashes/`.

Fuzzing can be very CPU intensive, so doing it on your laptop isn't the most efficient way to do it, a large cloud server can be a good choice for serious fuzzing.

And that's all there is to it! Happy Fuzzing!