freqens Documentation

Release 0.0.1

Braulio Valdivielso

October 09, 2015

Contents

Contents:

CHAPTER 1

Tutorial

In this basic tutorial, we'll see how we can use freqens to break some weak crypto.

1.1 Scenario

You are working for the NSA and find that some terrorists are using brainfuck programs to hack the whole universe. Some intelligence lets you know that they are using extremely weak crypto (single byte xor) in order to "secure" their communications, which you have been able to intercept.

In particular, you want to decode a ciphertext that looks like this:

```
5  555 P55'P5     7PP&5V77V7P5V5V5&P77     55&P77&&&&55&P&5V7VV\x017P7&7P7V 7 P5V77 5&555V7V7P7V5P&P5
777   5 5\x01++P&\x01++++P77 5&5&\x01++++++P77P&V55&\x01++++++++P77  5 5&\x01++++++++++P77&&5&55
7&\x01++++++++++++P77 5 55&&7&\x01+++++++++++++++P77&5&5&\x01++++++++++++++++P77    5 55 7&\x01++++++
1+++++++++++++++++++P77&5&5&\x01++++++++++++++++++++P77&55&\x01++++++++++++++++++++P77
555&7&\x01++++++++++++++++++++++++P77&&&&555  7&\x01++++++++++++++++++++++++++P77  555
7&\x01++++++++++++++++++++++++++++++P5P&V7&\x01++++++++++++++++++++++++++++++++P77&555
7&\x01++++++++++++++++++++++++++++++++++P77&555&&7&\x01++++++++++++++++++++++++++++++++++P77    5 5
7&\x01++++++++++++++++++++++++++++++++++++++P77P&V5&55  7&\x01++++++++++++++++++++++++++++++++++++++
55&&7&\x01++++++++++++++++++++++++++++++++++++++++P7&55
7\x01++++++++++++++++++++++++++++++++++++++++++++P77&55&\x01VVVVVVVVVVVVVVVVVVVVVVV7P&55P77 55&V777P55
777&VV55V\x015P&P&&&P&7VV5V5P   P7     5&&V5V 7  PP5     7&V7V55P&%5V
```

1.2 Getting an analyzer

An analyzer represents the ideal frequency distribution your target plaintext has. Once it has been fed, it can be asked to score strings based on how legit they seem (how similar its frequency distribution is to the analyzer's). There are several ways of building an analyzer.

- **From a raw file: you can build an analyzer like**

```
from freqens.analyzer import Analyzer

# a representative sample of the target frequency distribution
# ie. a normal brainfuck program
filename = "./program.bf"
analyzer = Analyzer.from_raw_file(filename)
```

- **From a frequency distribution file: which is a json file containing some absolute frequencies.**

An example of a frequency distribution file would be:

```
{
        "a": 4,
        "b": 3
}
```

This is how you build an analyzer from one of these files:

```
from freqens.analyzer import Analyzer

filename = "./bf-distribution.json"
analyzer = Analyzer.from_file(filename)
```

- **From a string:**

```
from freqens.analyzer import Analyzer

analyzer = Analyzer("representative text")
```

For this scenario, the easiest way to build the bf analyzer is to use the `freqens` command line utility which lets you extract a fr

```
$> cd my-bf-programs
$> freqens *.bf > bf_frequency_distribution.json
```

1.3 Breaking the code

Now that you know how to get a brainfuck analyzer, it's time to break the code. We'll decode the ciphertext with every possible key (as it is single byte xor, there's only 256 possible keys) and let the analyzer discover what is the real ciphertext. Our program will look like:

```
from freqens.analyzer import Analyzer

def single_byte_xor(text, byte):
        return "".join( chr(c ^ byte) for c in bytearray(text) )

with open("ciphertext.txt") as ciphertext_file:
        ciphertext = ciphertext_file.read()
        analyzer = Analyzer.from_file("bf_frequency_distribution.json")

        possible_plaintexts = ( single_byte_xor(ciphertext, byte) for byte in range(256) )

        answer = analyzer.choose_best(possible_plaintexts)

        print answer[0] # Solution !!!
```

And the program will print:

```
+++>++>>>+[>>,[>+++++<[[->]<<]<[>]>]>-[<<+++++>>-[<<----->>-[->]<]]
<[<--<[<]+<+[>]<<+>->>>]<]<[<]>[-[>++++++<-]>[<+>-]+<<<+++>+>
  [-
    [<<+>->-
      [<<[-]>>-
        [<<++>+>-
          [<<-->->>+++<-
```

```
              [<<+>+>>--<-
                [<<->->-
                  [<<++++>+>>+<-
                    [>-<-
                      [<<->->-
                        [<<->>-
                          [<<+++>>>-<-
                            [<<----->>>++<-
                              [<<++>>>+<-
                                [>[-]<-
                                  [<<->>>+++<-
                                    [<<->>>--<-
                                      [<<++++>+>>+<-
                                        [<<[-]>->>++<-
                                          [<<+++++>+>>--<-
                                            [<->>++<
                                              [<<->>-
]]]]]]]]]]]]]]]]]]]]]]]]]<[->>[<<+>>-]<<<[>>>+<<<-]<[>>>+<<<-]]>>]
>[-[---[-<]]>]>[+++[<+++++>--]>]+<++[[>+++++<-]<]>>[-.>]
```

Which is obviously an Universal Turing Machine! Now you know terrorists have turing-complete technology in their hands.

CHAPTER 2

Analyzer

This module contains two classes:

- Analyzer: a generic analyzer. It can be fed both from text strings and from files. You can also store a representation of the state of the analyzer to be retrieved later, with the `from_file` class method or the `load` method.
- EnglishAnalyzer: an special analyzer for the English language.

class `freqens.analyzer.`**Analyzer**(*content=None*)

The class that performs the analysis. You can feed an analyzer from different sources (strings, files...) so that it extracts the target frequency distribution and ask it to score supplied content based on frequency similarity

choose_best(*strings*, *n=1*)

Returns the n strings whose frequency distribution is most similar to the one fed to the analyzer.

Parameters

- **strings** – an iterator with the strings where the Analyzer will looked for the best strings.
- **n** – an integer specifying the number of strings which will be returned.

Returns an iterable containing the `n` best strings sorted by frequency similarity

discard(*chars*)

Removes the chars in chars from the counter

Parameters **chars** – an interable consisting of the chars whose frequency will be set to 0

feed(*content*)

Feeds the analyzer with a string

Parameters **content** – the string to be fed to the analyzer

feed_from_raw_file(*filename*)

Feeds the analyzer with the content of a file Every character will be taken into account, including newline chars.

Parameters **filename** – the path of the file that will be fed to the analyzer

classmethod **from_file**(*filename*)

Reads a frequency distribution from a JSON file as stored by store method

classmethod **from_raw_file**(*filename*)

Returns an analyzer whose frequency distribution is read from the file content

keys()

Returns the characters whose frequency is greater than 0

load(*filename*)
Loads a frequency distribution file and adds it to the current distribution

score(*content*)
Assigns a score to any string. The smaller, the more similar frequency distribution. 0 means that the frequency distributions of both the content and the analyzer are equal.

Parameters **content** – the string to be scored.

Returns a float number

serialize()
Returns a json representation of the analyzer

Returns a string containing a json representation of the absolute frequencies the analyzer has been fed with.

store(*filename*)
Stores the json representation of the analyzer to a file

transform_keys(*transformation*)
Maps the keys to other new keys to get a new frequency distribution

The relative frequency of keys that map to the same key will be added in order to get the new frequency distribution.

Parameters **transformation** – a callable object that maps chars to chars

class freqens.analyzer.**EnglishAnalyzer**(*blank_spaces=True*, *case_sensitive=True*, *just_alpha=False*)
An analyzer for the english language

freqens.analyzer.**counter_distance**(*counter1*, *counter2*)
Euclidean distance on the frequency distribution space

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

f

A

C

D

E

F

K

L

S

T