Repeated XOR - 70 (Cryptography)

Writeup by Gladius Maximus

Created: 2014-11-11 16:46:34

Last modified: 2014-11-17 20:58:07

Problem

There's a secret passcode hidden in the robot's "history of cryptography" module. But it's encrypted! Here it is, hex-encoded: encrypted.txt. Can you find the hidden passcode?

Hint

Like the title suggests, this is repeating-key XOR. You should try to find the length of the key - it's probably around 10 bytes long, maybe a little less or a little more.

Answer

Overview

Preform Kasiski elimination to find the key length. For this problem, the keylength is 8. Since the cipher text is encrypted using repeateing xor, you can do a frequency analysis on the 0th, 8th, 16th characters, and then another one on the 1st, 9th, and 17th characters, adn then on the 2nd, 10th, and 18th character.

Details

About repeated XOR

In a repeating XOR cipher, if the key is shorter than the message (it almost always is), the key is duplicated in order to cover the whole message. Then each byte of the plain text is XORed with each according byte of the key. For example, suppose we are trying to encrypt the message 'THIS IS A MESSAGE', with the key 'YOU', we first convert all of the characters to integers using ASCII encoding. (More examples may be found on the Wikipedia page)

```
 T   H   I   S       I   S       A       M   E   S   S   A   G   E
84, 72, 73, 83, 32, 73, 83, 32, 65, 32, 77, 69, 83, 83, 65, 71, 69
```

Then we duplicate the key and convert it to an integer sequence.

```
 Y   O   U   Y   O   U   Y   O   U   Y   O   U   Y   O   U   Y   O
89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79
```

Then we XOR the integers together.

```
84, 72, 73, 83, 32, 73, 83, 32, 65, 32, 77, 69, 83, 83, 65, 71, 69
89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79
xor -------------------------------------------------------------
13,  7, 28, 10,111, 28, 10,111, 20,121,  2, 16, 10, 28, 20, 30, 10
```

Do you notice the repeating 28, 10, 111? That is because there was a repeat in the paintext that was a multiple of the keylength apart. In reality, repeats happen quite frequently, but it is just as likely to get a repeat 3 characters apart as it is to get one 4 characters apart. If they are 4 characters apart, a repeat is not shown in the cipher text, because the length of the offset (4) is not a multiple of the key length (3). For example, encrypting 'THIS MESSAGE IS NOT' with the key 'YOU' gives:

```
84, 72, 73, 83, 32, 77, 69, 83, 83, 65, 71, 69, 32, 73, 83, 32, 78, 79, 84
89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79, 85, 89, 79, 85, 89
xor ----------------------------------------------------------------------
13,  7, 28, 10,111, 24, 28, 28,  6, 24,  8, 16,121,  6,  6,121,  1, 26, 13
```

XOR has a lot of special properties. First XOR is commutative. $$a \oplus b = b \oplus a$$ ($$\oplus$$ stands for XOR). It is associative $$a \oplus (b \oplus c) = (a \oplus b) \oplus c $$. Next, anything XORed with itself is zero: $$ a \oplus a = 0 $$, and anything XORed with zero is anything: $$ a \oplus 0 = a$$. Thus we can conclude that $$ a \oplus b \oplus b = a \oplus (b \oplus b) = a \oplus 0 = a$$. Furthermore, if $$a \oplus b = c$$, then $$a \oplus c = a \oplus (a \oplus b) = (a \oplus a) \oplus b = 0 \oplus b = b$$.

If that all went over your head, just know that XOR is its own inverse. That is the most important part of XOR and it will come in handy later.

However, because repeated text shows up frequently in the English language, we can expect that at least some of the time the repeated text will be offset by a multiple of the key. We can count on most of the repeated cipher text being an integer multiple of the key. Thus we can find the key length

First, I made encrypted.py which contains all of the raw data and parsed data. Parsing the data into different forms is usually a good place to start.

```python
from encrypted import enc_numbers

def shift(data, offset):
    return data[offset:] + data[:offset]

def count_same(a, b):
    count = 0
    for x, y in zip(a, b):
        if x == y:
            count += 1
    return count

print ('key lengths')
for key_len in range(1, 33): # try multiple key lengths
    freq = count_same(enc_numbers, shift(enc_numbers, key_len))

    print ('{0:< 3d} | {1:3d} |'.format(key_len, freq) + '=' * (freq / 4))
        # ^ this line does fancy formatting that outputs key_len and then freq and
        # then a bar graph
```

This makes a nice bar graph showing how many characters were the same after shifting the original cipher text over by `key_len` offset. You can see that 8, 16, and 32 have a much higher amount of same-characters. Thus, we can guess that the key is 8 characters long.

```python
key_len = 8

frequencies = []
for i in range(0, key_len):
    frequency = Counter()
    for ch in enc_ascii[i::key_len]:
        frequency[ch] += 1
    frequencies.append(frequency)
```

A snipped version of the output is below:

```
...
 3  |  12 |===
 4  |  18 |====
 5  |   9 |==
 6  |   2 |
 7  |  42 |==========
 8  | 249 |==============================================================
  9  |  64 |================
 10 |   2 |
 11 |   6 |=
...
```

Once we know the key length, each character and the character 8 down are XORed with the same key character. Thus we can do a frequency analysis. Imagin splitting up the message into rows 8 characters long. Each column was encrypted with the same character of the key, so each column should have character frequencies that correspond. Thus we can go through each 0th, 8th, 16th, ... character and pick the most frequently occuring character and then do the same for the 1st, 9th, 17th, etc.

```
from collections import Counter

key_len = 8
frequencies = []
for i in range(0, key_len):
    frequency = Counter()
    for ch in enc_ascii[i::key_len]:
        frequency[ch] += 1
    frequencies.append(frequency)
```

Once we have the most frequently occuring character, we say it decrypts to ' ' (space occurs most frequently in a block of text). Let the most frequently occuring character in the nth column of the cipher text $$c_n$$. Let the nth character of the key is $$k_n$$. Let ' ' be $$m$$. We know that $$m \oplus k_n = c_n$$. XOR is its own inverse, so $$m \oplus c_n = k_n$$. So because XOR is its own inverse, we can find the key by XORing cipher text and known plain text Thus we can find the key if we know the most common character in english and the most common character in the nth column.

We can check this by printing out what other common characters decrypt to. If we have the right answer, we should get other common characters decrypting to t, e, a, i or something like that.

```
print ('guesses for most common letters')
key_numbers = []
for frequency in frequencies:
    k = ord(frequency.most_common(1)[0][0]) ^ ord(' ')
    print ('{k} -> \' \''.format(**locals()))
    key_numbers.append(k)

    others = ''
    for val, freq in frequency.most_common(10):
        others += chr(ord(val) ^ k) + ' '
    print ('Other common letters: {others}\n'.format(**locals()))
```

A snipped version of the output is below. It looks like it worked.

```
...
34 -> ' '
  e t a o r h n s i

82 -> ' '
  e t i o n a s l r

155 -> ' '
  e t o n d i r s a
```

We are ready to decrypt the whole text.

```
from itertools import izip, cycle
from encrypted import enc_numbers

def decrypt(c_num, k_num):
    return ''.join(chr(c ^ k) for c, k in izip(c_num, cycle(k_num)))

print ('decrypting text')
print (decrypt(enc_numbers, key_numbers))
```

I have attached the complete summarized script containing all of the parts of this writeup here (kasiski.py and its dependency encrypted.py)

Flag

```
89ae5d8b68c8d6323c79e2f8540f0f50728cffb7
```