|  |  |  |
| --- | --- | --- |
| Ebola Writeup | | |
|  |  |  |
|  | | |

# Description:

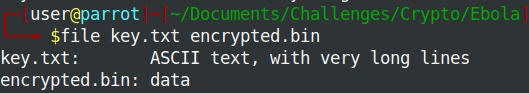
We suspect that some terrorists have a plan to use the Ebola virus. We have managed to collect an encypted message and its key. Can you help us decrypt the message?

**Skills Required:**

* Scripting
* Cryptanalysis
  + Frequency Analysis
  + digrams, trigram, quadgram… etc.
  + Alphabetic Ciphers (Monoalphabetic vs Polyalphabetic)
  + Word Patterns

# Solving the Challenge:

In this challenge we are given 2 files to start with, ‘encrypted.bin’ and ‘key.txt’. ‘Encrypted.bin’ is a binary file consisting of UTF garbage, and ‘key.txt’ is ASCII data.



**Encrypted.bin:**

**A picture containing text, blackboard

Description automatically generated**

**A picture containing bottle, photo

Description automatically generatedKey.txt:**

Since the extension of the encrypted file is .bin, and the data seems to be UTF interpreted junk, we know we’re going to be working with binary input, so we can prep a parser to interpret our input.

A close up of a screen

Description automatically generated

This is the standard loop I use to interpret binary data. It reads the first byte of the file using ‘enc.read(1)’, enters a while loop that loops until the byte variable is empty. Inside of this loop we are manipulating the byte a little bit to suit our specific needs.

* byte = enc.read(1) // byte = b‘\xda’
* ord(byte) // = 218
* bin(byte) // = ‘0b11011010’
* str(byte) // = “0b11011010”
* byte[2:] // = “11011010”
* rjust(8,’0’) // = pad the string the zeroes, up to 8 spaces

This gives us the encrypted.bin file in a nice formatted binary string that we can iterate through to test different decryptions. With the initial setup out of the way, we can move onto manipulating the data. Given the ‘key.txt’ file, it is clear to see that this is a DNA sequence, so my first though is some sort of DNA decryption, most likely a form of XOR decryption.

A black sign with white text

Description automatically generatedA screen shot of a clock

Description automatically generatedA quick google search of DNA encryption reveals the following table for converting a DNA sequence to binary bits. Using this we can convert the DNA key to binary, and with the already converted binary message XOR the two together to potentially receive our decrypted message.

A quick function to generate our binary XOR key, we read the file, convert each character to its binary representation according to our conversion table, and then append that binary to a string.

A screenshot of a cell phone

Description automatically generatedWe quickly run into a problem in doing this though, the length of our message in binary is 16088, and the length of our key in binary is 2048, so we will need to pad the key. In general, for XOR encryption with unequal length key and ciphertext, you round the key on itself to match the length.

The first line of our decryption function rounds the key to match the length of our ciphertext.

key \* (floor(len(message)) / len(key))

First we multiply the by the floored result of the length of the message divided by the length of the key. In python, multiplying a string by a number will concatenate the string with itself that number of times. In our example this will be

(16088 / 2048) = 7.8554..

floor(7.8554..) = 7

So now the length of our key will be 2048 \* 7 = 14336, but for XOR encryption we need an exact length match, so we need to add the remainder.

key[:len(message) % len(key)]

the remainder in our example is 1752, and the [:] is the python slice operator, so we are taking the contents of the string from 0 to 1752, and appending that to the already multiplied key, giving us an exact match. Now that we have our rounded key and binary input, we simply iterate through the key, and take each bit of the key, and XOR it with the corresponding bit from the binary input. Running this function gives us the decrypted message resulting in…

A close up of a screen

Description automatically generatedWell that’s definitely not right… after a little trial and error of swapping around the conversion tables, and reverse padding the key to no avail, I gave up on this method of decryption. Without being able to utilize the given key, we essentially start from zero in decrypting this message, so we’ll start with some common cryptanalysis techniques to gain more information on the ciphertext.

If we assume that the original plaintext of the message is English, and that the method of encryption uses a monoalphabetic cipher (1-to-1), then we can apply frequency analysis. The most commonly occurring English characters are

e t a o i n s r h l d c m u f p g w y b v k x j q z

So we can use these to attempt to generate an alphabet from the encrypted text.

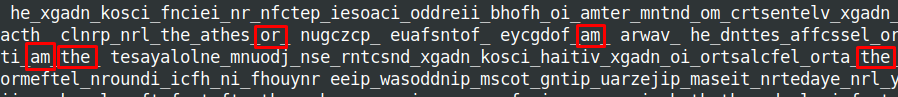
A close up of a black screen with text

Description automatically generatedThe first thing we’ll do is take each byte of the input, and each time that byte occurs in the file, we’ll increment the Counter object. After reading in the file, we loop through the 27 most commonly occurring characters, and build a conversion table mapping those characters, directly to the English letter frequency.

A picture containing wall, indoor

Description automatically generatedA close up of a screen

Description automatically generatedThe result is a dictionary with the decimal equivalent of each byte, mapped to an English letter. We can attempt to apply this directly to the ciphertext and decrypt it, however frequency analysis typically only works with the very frequently occurring characters (e,t,a…).

This function will loop through each character in the encrypted message and attempt to convert that character through our mapping. If the character is not in our mapping, a \_ is inserted instead of erroring, this will show us what characters we are missing in our conversion table.

A close up of a street

Description automatically generatedThe result of applying this conversion is gibberish, however we can start to see some potential words appearing throughout the text (the, or, am). There is no guarantee that these words are right, most of the time, words that appear through frequency analysis are pure coincidence. Despite this, we will continue decryption under the assumption that our conversion table is correct, and continuously modify it until we begin to see results.

Another important method of cryptanalysis is n-gram analysis. This is taking n-pair of words and finding the most frequently occurring pairs and matching them to the most frequently occurring pairs in the English language. From the image above, we can see that our most frequently occurring n-grams seem to be matching some common English n-grams (the, or, th, nt). Now that we have some knowledge on what the decryption alphabet could be, we can begin to decrypt the message using only the characters we think are correct, and an unknown character for the rest. This is a very trial and error process, and requires a lot of tweaking, and can be very time consuming. If you assume a set of characters in the beginning are correct, and these characters turn out to be incorrect, the entire alphabet you’ve built based on these characters will be wrong and you lose all progress.

A picture containing outdoor

Description automatically generatedUsing this method, we can iteratively build half-way decrypted plaintexts like this, and pick and choose what words or characters we find. A useful cryptanalysis method for this is word pattern analysis. For example, if our ciphertext word is ‘MLMLA’ we can apply word pattern analysis.

MLMLA -> 0.1.0.1.2

So regardless of which characters correspond to our numbers, we know the word must follow this pattern of repeating characters. Using a word pattern database, we know that the word corresponding to the pattern must be one of the following:

0.1.0.1.2

COCOA

MIMIC

PAPAL

VIVID

A picture containing wall

Description automatically generatedUtilizing these techniques we can build our alphabet, and after a lot of trial and error, the final decryption was revealed, along with the flag.