

Alexander Lotero
Offensive Security
Spring 2023

Week 12 - Crypto 1

This week's challenges were "SBXOR," "MBXOR," and "Linear." For credit I completed "SBXOR" and "MBXOR."

SBXOR

The first challenge that I completed this week was "SBXOR." The challenge provides us with a prompt - "a simple XOR. Can you crack it?" and file *ciphertext.txt*. As expected, downloading and opening the file provides us with a line of ciphertext that appears to be in hexadecimal format:

```
"fff5f8fee2dbebecedaac6ffa9ebfafcc6dfcdcec6aeaba1f8a0faaba0ffaef8fbe4"
```

Given that the challenge hint stated that this "encryption" was performed using XOR (1), and given the name of two of the challenges this week: "SBXOR" - single-byte XOR and "MBXOR" - multi-byte XOR, we make the correct assumption that this ciphertext was encrypted using a single-byte key. We draw from this week's slides, specifically the "XOR [cont]" slides, to build a python solver script *sbxorsolver.py*. Because this ciphertext was encrypted using a single-byte key, and because we know the format that plaintext flag will appear in: "flag{...", it is computationally reasonable to brute-force the key by trying every possible byte until one produces the "flag{..." string.

We start by reading the contents of the ciphertext file into a variable *ciphertext* - a string type of hex characters. To proceed we must perform some conversion. First, we convert the hex string into a list of hex strings where each element is a single byte using the following list comprehension:

```
cipher_bytes = [ciphertext[i:i+n] for i in range(0, len(ciphertext), n)]
```

Next, because we will be performing the XOR operation, we must convert each of these bytes to their corresponding integer value like so:

```
cipher_ints = []  
for m in cipher_bytes:  
    cipher_ints.append(int(m, 16))
```

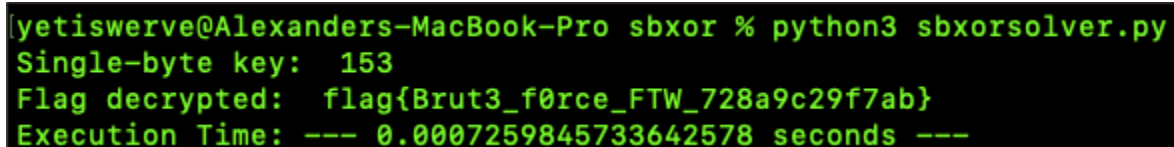
Finally, with our list of integers representing the original ciphertext - *cipher_ints*, we call

our *decryptor()* function:

```
def decryptor(cipher_ints, start_time):
    for i in range(0x00, 0xff):
        decrypted = ""
        for j in cipher_ints:
            decrypted += chr(i^j)
        if "flag{" in decrypted:
            print("Single-byte key: ", i)
            print("Flag decrypted: ", decrypted)
            print("Exec time: %s sec" % (time.time()-start_time))
            exit()
```

Here we loop through each possible single-byte key from 0x00 to 0xff. With each iteration, we XOR the byte from the ciphertext with the current key, convert that value back to a character string, and combine them all into a single string. From there, we check if our known first five characters, "flag{", are present in the string. If so, we print the single-byte key that was identified, the decrypted text (our flag), the program's execution time, and finally, the program exits.

We run our script within a terminal which provides us with the following output:

A terminal window with a black background and green text. The prompt is 'yetiswerve@Alexanders-MacBook-Pro sbxor %'. The command 'python3 sbxorsolver.py' has been executed. The output is: 'Single-byte key: 153', 'Flag decrypted: flag{Brut3_f0rce_FTW_728a9c29f7ab}', and 'Execution Time: --- 0.0007259845733642578 seconds ---'.

```
yetiswerve@Alexanders-MacBook-Pro sbxor % python3 sbxorsolver.py
Single-byte key: 153
Flag decrypted: flag{Brut3_f0rce_FTW_728a9c29f7ab}
Execution Time: --- 0.0007259845733642578 seconds ---
```

As shown in the image above, our script identified the key 153, 0x99 in hex, as well as our decrypted flag, indicating that we had successfully completed the challenge.

MBXOR

The second challenge I completed this week was "MBXOR" or "multi-byte XOR." Like "SBXOR" before it, we are provided with a hint - in this case indication that we are dealing with a multi-byte key, and a ciphertext file - *ciphertext.txt*. Again, we start by opening the ciphertext within a text editor, but in this case, unlike "SBXOR," the ciphertext is tens-of-thousands of characters long. This doesn't tell us much now, other than the fact that there is much more encrypted text than just our flag.

Our first step towards decryption, as stated in the slides, is identifying the length

of the xor encryption key. Note, we initially began working on a python script to programmatically predict the key length by identifying the “index of coincidence” (IOC) and the “hamming distance” (2, 3), as recommended by the slides. We got as far as what we believe was a working hamming distance solver and a nearly working IOC solver. However, while researching and troubleshooting our methods to do so, we came across a publicly available command line tool for xor decryption analysis, “xortool.” Specifically, the tool is for: “guessing key length based on the count of equal characters, and guessing the key itself based on knowledge of the most common character in the plaintext” (4). We install the tool via its Github repository as well as any additional dependencies with *pip install*. After reading through the tool’s help page (*xortool - h*), we split our problem into two smaller problems and create two separate solver scripts to address them - both of which are available with this submission.

Our first problem, predicting the key length, we approach with a python script, *mbxorkeylength.py*. Utilizing the *subprocess* library, we create a terminal process to run *xortool* on our ciphertext and store the command’s output into a variable *key_lengths*:

```
# run xortool and store the output
key_lengths = subprocess.getoutput('xortool ciphertext.txt')
```

The traditional terminal output of the command appears as follows:

```
yetiswerve@Alexanders-MacBook-Pro mbxor % xortool ciphertext.txt
The most probable key lengths:
 2: 13.9%
 5: 17.1%
10: 16.8%
15: 10.6%
18:  6.6%
20: 10.5%
22:  5.6%
25:  6.8%
30:  7.1%
40:  5.1%
Key-length can be 5*n
Most possible char is needed to guess the key!
```

Because we have assigned this output to a variable, the next several steps/lines of our code serve to process this output and filter out the extra information. This process leaves us with a single dictionary object *key_len_dict* in which the keys are the

estimated key length for the ciphertext, and the values are the likelihood of that key length as estimated by *xortool*. Finally, using this dictionary, we identify the key length with the highest reported likelihood and print that length to our terminal:

```
yetiswerve@Alexanders-MacBook-Pro mbxor % python3 mbxorkeylength.py  
xortools identified the following flag length: 5
```

As seen above, our script reports that *xortool* identified the key length of 5 to be most likely for the provided ciphertext.

The second problem that we approach is using this estimated key length to brute force the xor key and convert our ciphertext back to plaintext. In this step, we switch to using a bash script *mbxorsolver.sh*. Here, we ultimately run three bash commands to print our flag. First, we run the following *xortool* command:

```
xortool -x -l 5 -c " " ciphertext.txt
```

Our command makes use of the following options (4):

1. `-x`: to indicate that our provided ciphertext is a hex-encoded string
2. `-l 5`: to indicate the key length as calculated in the previous step
3. `-c " "`: to indicate that the blank space character is the most common character found within the plaintext. We included this option because *xortool* requires that the most common character is included to help the brute forcing process. Further, we selected the blank space character specifically because given the length of the text, tens-of-thousands of characters, we assume that the additional noise, beyond our flag, is some copy-and-pasted english text. If true, the blank space will occur more than any other single character.

Next, we use the *echo* command to print a blank line for readability's sake, and print the "Flag found: " string to be followed by the flag. Finally, to find the flag, we use the *cat* command to print the contents of the *0.out* - the decrypted text file created by *xortool* in its newly created *xortool_out* folder. Further, because the text is too long to print to a single screen, we pipe the output of the *cat* command to a *grep* command to only print the line including the "flag{" substring:

```
# find the flag within our full decrypted text  
cat xortool_out/0.out | grep "flag{"
```

Running this script, we receive the following:

```
[yetiswerve@Alexanders-MacBook-Pro mbxor % sh mbxorsolver.sh
1 possible key(s) of length 5:
nSXVt
Found 0 plaintexts with 95%+ valid characters
See files filename-key.csv, filename-char_used-perc_valid.csv

Flag found:
flag{n0t_s0_b@d_1s_it?_563ef5305b88}
```

As seen in the image above, the *xortool* command brute forced the key and the decryption process, needing only the length and most common character. The output reveals our flag, indicating that we had successfully completed the challenge.

References

1. <https://mathworld.wolfram.com/XOR.html#:~:text=A%20connective%20in%20logic%20known,539%20and%20550%2D554>)
2. https://en.wikipedia.org/wiki/Index_of_coincidence
3. <https://www.geeksforgeeks.org/hamming-distance-two-strings/>
4. <https://github.com/hellman/xortool>
- 5.