Assignment 2: OTP

Command Usage:

`otp.py [TASK]... [FILES]...`

Task Flags:

`-1 ...`

`-2 INFILE OUTFILE`

`-3 IN.bmp OUT.bmp`

`-4 IN1.bmp IN2.bmp OUT.bmp`

Repeating Logic:

1) **`xor_otp()`**

   All tasks used this function.

   Checked to see if the encryption key and text were of the same length. Then xor-ing the bytes between them.

```
def xor_otp(key: bytes, text: bytes):

    if(len(key) == len(text)):
        encrypted = xor_bytes(key, text)
    else:
        raise Exception("Key and text are not the same size!")

    return encrypted
```

2) **`xor_bytes()`**

   This xor's the contenst of two bytes objects.

```
def xor_bytes(str1, str2):
    return bytes(a ^ b for a, b in zip(str1, str2))
```

3) **random_bytes()**

This reads random bytes from `/dev/urandom` for a given size.

This is used in most tasks to produce a random key.

```
def random_bytes(size: int):
    return open("/dev/urandom", "rb").read(size)
```

4) **checkDecryption()**

This is used in some tasks to validate to the user that two byte_strings actually have the same value.

```
def checkDecryption(str1, str2):
    isCorrect = False
    if(str1 == str2):
        isCorrect = True
    print("Valid Decryption: ", isCorrect)
```

Tasks:

1) `./otp.py -1`

This task only tests to see if the xor functionality is working between byte strings.
My only difficulty was removing TypeErrors.

```
def task1():
    encrypted = xor_otp(b'Darlin dont you go',
            b'and cut your hair!')
    print(encrypted.hex())
```

The output was just the hex dump of the xor contents.

Printing:

```
250f164c0a1b54441601015259071449154e
```

2) ./otp.py -2 ./files/in.txt out.txt

This task was used to encrypt entire files, overwriting header information as well as its contents. This required the file to be opened as a byte reader, which took a little time to figure out.

```
def task2():
    infile = sys.argv[2]
    outfile = sys.argv[3]

    # Encrypt text
    text = open(infile, 'rb').read()
    key = random_bytes(len(text))
    encrypted = xor_otp(text, key)

    # Write encryption to file
    open(outfile, "wb").write(encrypted)

    # See if decryption is correct
    decrypted = xor_bytes(encrypted, key)
    checkDecryption(text, decrypted)
```

The output for this task is hard to display, because most of the output characters are unprintable. However within the program there is a function that checks to see if the decryption was done correctly by decrypting the encrypted text and comparing it to the original text.
This function would print:

```
Valid Decryption:  True
```

When decryption was reversible.

3) ./otp.py -3 ./files/mustang.bmp task3out.bmp

This task added the difficulty of dealing with a byte-offset where the certain header information, for a bmp file, would not be encrypted. Otherwise, the encryption was the same.

```
def task3():
    # Modify image file with 54 byte header

    infile = sys.argv[2]
    outfile = sys.argv[3]

    file_bytes = open(infile, 'rb').read()

    # Separate header from text
    splitat = 54
    header, text = file_bytes[:splitat], file_bytes[splitat:]

    # Encrypt text
    key = random_bytes(len(text))

    encrypted = xor_otp(text, key)

    # Write encryption to file
    open(outfile, "wb").write(header + encrypted)

    # See if decryption is correct
    decrypted = xor_otp(encrypted, key)
    checkDecryption(text, decrypted)
```

Here is the output file created from this encryption:

Figure 1: Resulting encryption bmp file

View the images. What do you observe? Are you able to derive any useful information about from either of the encrypted images? What are the causes for what you observe?

From this image, I could see absolutely no pattern to its output. The colors have very little association to each other and have wide distribution. What caused this randomness was purely caused by the randomness of the key, which was read from `/dev/urandom`. This key being completely random, overshadowed the file contents which did have a pattern.

4) `./otp.py -4 ./files/mustang.bmp ./files/cp-logo.bmp task4out.bmp`

This task required two files to be encrypted with the same key. After this, the two files would be xor'ed to each other, which produced a file that was hardly encrypted, and the contents of both files were easily determined. The purpose of this task was to display the importance of not re using keys for encryption since the files then have an easy was to be decrypted.

```
def task4():
    infile1 = sys.argv[2]
    infile2 = sys.argv[3]
    outfile = sys.argv[4]

    file_bytes1 = open(infile1, 'rb').read()
    file_bytes2 = open(infile2, 'rb').read()

    # Separate header from text
    splitat = 54
    header = file_bytes1[:splitat]
    text1 = file_bytes1[splitat:]
    text2 = file_bytes2[splitat:]

    # Encrypt text
    key = random_bytes(len(text1))

    encrypted1 = xor_otp(text1, key)
    encrypted2 = xor_otp(text2, key)

    # Try revert, by xor-ing encryptions
    try_decrypt = xor_bytes(encrypted1, encrypted2)

    # Write xor decryption to file
    open(outfile, "wb").write(header + try_decrypt)
```

Here is the output file created xor'ing the two file encryptions:

Figure 2: bmp file of xor between encrypted files w/ shared key

View the output. Are you able to now derive any useful information about the original plaintexts from the resulting image? What are the causes for what you observe?

Now looking at the output, we can see extremely useful information regarding the file contents. In fact, we can determine the contents of both files simply by looking at it, that being a mustang and the cal poly logo. This is because an xor operation is a reversible action if you know certain enough information. In this case we know that the same key was used twice on both files, meaning an xor between the two files would provide an unencrypted file with the original files xor'ed together.

Conclusion:

The purpose of this lab was to display how important the use of unique keys are to an encryption possess, especially an otp encryption. This is displayed in how easy the breaking of the encryption was, simply by xor'ing files with non-unique keys. From this, I can see that modern security solutions to encryption may be far more complicated than a simple xor to a string.