

Secret Phrase:

Looking at the ciphertext, I knew it could not be a substitution cipher as those were letter only. This left me with either XOR or Caesar cipher.

Given that for XOR it was paramount to have at least a sliver of knowledge about the plaintext, therefore, it was unlikely to be a XOR cipher.

For Caesar, I knew that ASCII could push letters such as "DEL" which was part of the ciphertext.

With this thought, I made a small script that would shift the ciphertext through all possible lengths.

```
with open("../Course-CTF1/CTF-Crypto/challenge1.txt", "r") as file:
    ciphertext = file.read()
    print(type(ciphertext))

def ascii_shift(ciphertext, shift_value):
    decrypted_text = ""
    for char in ciphertext:
        # Convert character to ASCII, apply shift, and convert back to character
        decrypted_text += chr((ord(char)-shift_value)%128)
    return decrypted_text

for shiftlength in range(1, 128): # Shifting through ASCII bounds
    decrypttext = ascii_shift(ciphertext, shiftlength)
    print(f"Shift {shiftlength}: {decrypttext}")
```

This ultimately gave me more than 100 lines to sift through and legible English words. The flag was found on Shift 11.

```
Shift 10: Ti2guDszqu1jtXf5lDszqu1
Shift 11: Sh1ftCrypt0isWe4kCrypt0
Shift 12: Rg0esBqxos/hrVd3jBqxos/~
```



Secret Text:

For this ciphertext, it was evident that either a Caesar or substitution cipher was used since the text contained only letters.
Given the context of this CTF challenge and the fact that the first ciphertext used a Caesar cipher, I suspected this one might be encrypted using a substitution cipher instead.

To begin, I ran the entire ciphertext through a letter frequency analyzer. This approach gave me a solid starting point for mapping the substitution, as the most frequent letters in the ciphertext likely correspond to the most common letters in English (e.g., E, T, A, etc.).

Letter Frequency Counter

JHYWNP
HMS
JW
JHQ
G
CYD
IDT
CKG
JWYYQKJMCZ
YCMK
WNJIMGQ.
"TWN
ZWWD
JYWNUZQG,
TWNKP
OWJJQY,"
ICMG
KMBD,
RWZGMKP
C
JYCKIOCYQKJ
ZQJJQY
CI
HQ
IOWDQ
CKG
JNBDMKP
MJ
MKIMGQ
HMI
GWNUZQJ.
"IW
GW
TWN,"
ICMG
HCYYT.
"CH,"
KQCYZT
HQCZQII
KMBD
XCEQG
CK
QZQPKJ
HCKG,
"C
SCJJQY
WR
KW
MSOWYJCKBQ.
. . .
MJ'I
KWJ
CI
JHWNPH
M
YQCZZT
XCKJQG
JW
LWMK.
. . .
JHWNPHJ
M'G
C00ZT,
UNJ
C00CYQKJZT
M
'GWK'J
RNZRMZZ
YQVNMYSQKJI'
-"MK
IOMJQ
WR
HMI
CMYT
JWKQ,
JHQYQ
XCI
C
ZWWD
WR
PYQCJ
UMJJQYKQII
WK
HMI
RCBQ."
UNJ
TWN
XWNZG
JHMKD,
XWNZGK'J
TWN,"
HQ
QYNOJQG
INGGQKZT,
ONZZMKP
JHQ
ZQJJQY
UCBD
WNJ
WR
HMI
OWBDQJ,
"JHCJ
IJYWKPIJ
OCIIWYG
QEYQ
GQEMIQG
MI
JHMIOCIIXWYGMIEQYTIQBNYQ?""WH
-
TQI,"
ICMG
HCYYT,
XHW
XCI
WUEMWNIZT
IN00WIIQG
JW
CPYQQ.



Letters: 2007

Copy Text

Letter Frequency Displayed Below:

Sort

Highest ▾

☐ Include Uppercase and Lowercase Separately

Letter	Occurrences	Percentage
Q	236	11.76%
J	160	7.97%
C	156	7.77%
W	154	7.67%
I	151	7.52%
Y	139	6.93%
K	136	6.78%

To refine the substitution mapping, I wrote a script that allowed me to visualize the changes in the ciphertext as I adjusted the letter mappings.

```
with open("../Course-CTF1/CTF-Crypto/challenge2.txt", "r") as file:
    ciphertext = file.read()

def substitution_decrypt(ciphertext, mapping):
    # Create a translation table based on the mapping
    table = str.maketrans(mapping)
    return ciphertext.translate(table)

# Define the initial letter mapping (Ciphertext -> English)
letter_mapping = {
    'A': 'Z',
    'B': 'C',
    'C': 'A',
    'D': 'K',
    'E': 'V',
    'F': 'X',
    'G': 'D',
    'H': 'H',
    'I': 'S',
    'J': 'T',
    'K': 'N',
    'L': 'J',
    'M': 'I',
    'N': 'U',
    'O': 'P',
    'P': 'G',
    'Q': 'E',
    'R': 'F',
    'S': 'M',
    'T': 'Y',
    'U': 'B',
    'V': 'Q',
    'W': 'O',
    'X': 'W',
    'Y': 'R',
    'Z': 'L'
}

plaintext = substitution_decrypt(ciphertext, letter_mapping)
print("Partially decrypted text:")
print(plaintext)
```

Gradually, I used common English language patterns to decipher the text. I uncovered a reference to "Nearly Headless Nick" from the Harry Potter series along the way.

AS HE WAS. NEARLY HEADLESS NIBJ, THE GHOST OF GRIZZLE
G UNDER HIS MREATH, ". . . DON'T ZULZILL THEIR REVUI
AID HARRY."HELLO, HELLO," SAID NEARLY HEADLESS NIBJ,
ON HIS LONG BURLY HATR, AND A TUNTB WITH A BUZZ, WH

After some Google searches and inferences, I located the flag.

NCE. . . . IT'S NOT AS THOUGH I REALLY WANTED TO JOIN. . . . THOUGHT I'D APPET,
/ TONE, THERE WAS A LOOK OF GREAT BITTERNESS ON HIS FACE."BUT YOU WOULD THINK, WO
S POCKET, "THAT STRONGEST PASSWORD EVER DEvised IS THISPASSWORDISVERYSECURE?"OH

Secret Zip File:

Given the length of this ciphertext, it made more sense to consider a block or stream cipher as the likely method.

Since the CTF prompt specifically mentioned that the encrypted zip file contained information about Scrooge McDuck's safe password, I suspected XOR encryption might be in play.

I suspected XOR because a common flaw of XOR algorithms is that, if you know some of the plaintext, you can gain a partial key.

In this case, I leveraged the common header of a zip file to extract a portion of the XOR key. All inputs were converted to hex format, and outputs to ASCII for readability.

```
# Defined the key in hex format
key = [0x4C, 0x50, 0x43, 0x45, 0x52, 0x30]

#zipfile_header = [0x50, 0x4B, 0x03, 0x04] -> [0x4C, 0x50, 0x43, 0x45]

# XOR decryption function
def xor_decrypt(ciphertext, key):
    decrypted_text = bytearray()
    key_length = len(key)

    for i in range(len(ciphertext)):
        # XOR each byte of the ciphertext with the corresponding byte of the key
        decrypted_byte = ciphertext[i] ^ key[i % key_length]
        decrypted_text.append(decrypted_byte)

    return decrypted_text

# Function to convert decrypted content to hex and human-readable form
def format_decrypted_content(decrypted_data):
    hex_output = decrypted_data.hex()
    formatted_hex = ' '.join(hex_output[i:i+2] for i in range(0, len(hex_output), 2))
    # Add spaces between pairs

    readable_output = ''.join([chr(byte) if 32 <= byte < 127 else '.' for byte in
decrypted_data]) # Human-readable characters

    result = []
    # Display in chunks of 32 hex characters (16 bytes)
    for i in range(0, len(formatted_hex), 48): # 48 characters = 16 bytes in hex (with
spaces)
```

```

        chunk_hex = formatted_hex[i:i+48]
        chunk_readable = readable_output[i//3:(i//3)+16] # Modified for readability
purposes
        result.append(f"{chunk_hex} | {chunk_readable}")

    return "\n".join(result)

with open("../Course-CTF1/CTF-Crypto/challenge3", "rb") as file:
    ciphertext = file.read()

# Decrypt the ciphertext using XOR and the provided key
decrypted_data = xor_decrypt(ciphertext, key)
formatted_output = format_decrypted_content(decrypted_data)

output_file_path = "../Course-CTF1/CTF-Crypto/decrypted_output.txt"
with open(output_file_path, "w") as output_file:
    output_file.write(formatted_output)

```

By XORing the ciphertext with the known plaintext (the zip file header), I was able to recover a partial section of the XOR key.

```

4c 50 43 45 08 7b 4f 54 13 0e 47 75 32 42 40 41 | LPCE.

```

After using this partial key to decrypt the ciphertext, I began noticing patterns in the decrypted text—specifically, what appeared to be part of the word "secret."

Given that "cret" appeared multiple times, I inferred that it was no coincidence and expanded the key to reveal the full word "secret."

```

1e 50 43 d1 9f de 8c 15 11 36 37 63 | ...u.PC.....67c
2e 74 1f 3f 5c 61 7e 07 7b 05 17 70 | ret/.t.?\a~.{..p

```

This led me to search the decrypted text for more patterns, focusing on key words from the CTF prompt, like "safe" and "Scrooge."

Eventually, I found the words "safe" and "pass," which confirmed the correct key length.

With this, I was able to decrypt the text fully and obtain the password.

```

3 6f 7c e5 66 71 7c e5 66 75 78 0b 00 | T...o|.fq|.fux..
1 00 00 04 14 00 00 00 54 68 65 20 70 | .....The p
3 6f 64 65 20 66 6f 72 20 74 68 65 20 | asscode for the
5 20 69 73 20 33 39 32 30 31 38 37 32 | safe is 39201872
1 32 32 33 30 0a 50 4b 01 02 1e 03 0a | 31812230.PK.....

```

Secret Photo:

When I first encountered this encrypted image, I initially thought it might have been encrypted using XOR, as image files typically contain a header that can be easily generated. The prompt also provided a clue, mentioning a vintage Olympus 1100 camera. However, after trying various image headers, I realized XOR was not the right approach.

I then pivoted to block ciphers, specifically considering that the image could have been encrypted using an ECB (Electronic Codebook) block cipher. ECB is known for a security flaw when encrypting large amounts of data, which can leave noticeable patterns in the ciphertext.

```
def format_decrypted_content(decrypted_data):
    hex_output = decrypted_data.hex()
    formatted_hex = ' '.join(hex_output[i:i+2] for i in range(0, len(hex_output), 2))
    # Add spaces between pairs

    readable_output = ''.join([chr(byte) if 32 <= byte < 127 else '.' for byte in
decrypted_data]) # Human-readable characters

    result = []
    # Display in chunks of 32 hex characters (16 bytes)
    for i in range(0, len(formatted_hex), 48): # 48 characters = 16 bytes in hex (with
spaces)
        chunk_hex = formatted_hex[i:i+48]
        chunk_readable = readable_output[i//3:(i//3)+16] # Modified for readability
purposes
        result.append(f"{chunk_hex} | {chunk_readable}")

    return "\n".join(result)

with open("../Course-CTF1/CTF-Crypto/challenge4", "rb") as file:
    ciphertext = file.read()

formatted_output = format_decrypted_content(ciphertext)

output_file_path = "../Course-CTF1/CTF-Crypto/challenge4.txt"
with open(output_file_path, "w") as output_file:
    output_file.write(formatted_output)
```

The signs of ECB encryption were evident when I converted the encrypted image into hex format, as I could clearly see repeated patterns within the ciphertext.

4	e7 d7 0e 2e f8 31 a5 1d 6a ef 49 0a 59 f2 9a 0e	1.
5	73 f6 bc 1c 9c 99 ae 28 73 02 dd 57 15 da d9 99		s.....
6	73 f6 bc 1c 9c 99 ae 28 73 02 dd 57 15 da d9 99		s.....
7	73 f6 bc 1c 9c 99 ae 28 73 02 dd 57 15 da d9 99		s.....
8	73 f6 bc 1c 9c 99 ae 28 73 02 dd 57 15 da d9 99		s.....
9	73 f6 bc 1c 9c 99 ae 28 73 02 dd 57 15 da d9 99		s.....
10	73 f6 bc 1c 9c 99 ae 28 73 02 dd 57 15 da d9 99		s.....
11	73 f6 bc 1c 9c 99 ae 28 73 02 dd 57 15 da d9 99		s.....

If the image was indeed encrypted using ECB, there was no need to fully decrypt it, since the patterns would still be visible to the human eye despite the encryption. I decided to replace the encrypted image's header with a new one that matched the size and constraints of a vintage Olympus 1100 camera.

After several attempts with different image headers, I eventually succeeded in viewing the encrypted image by using a PPM header.

