RazorHack 2024 Writeup

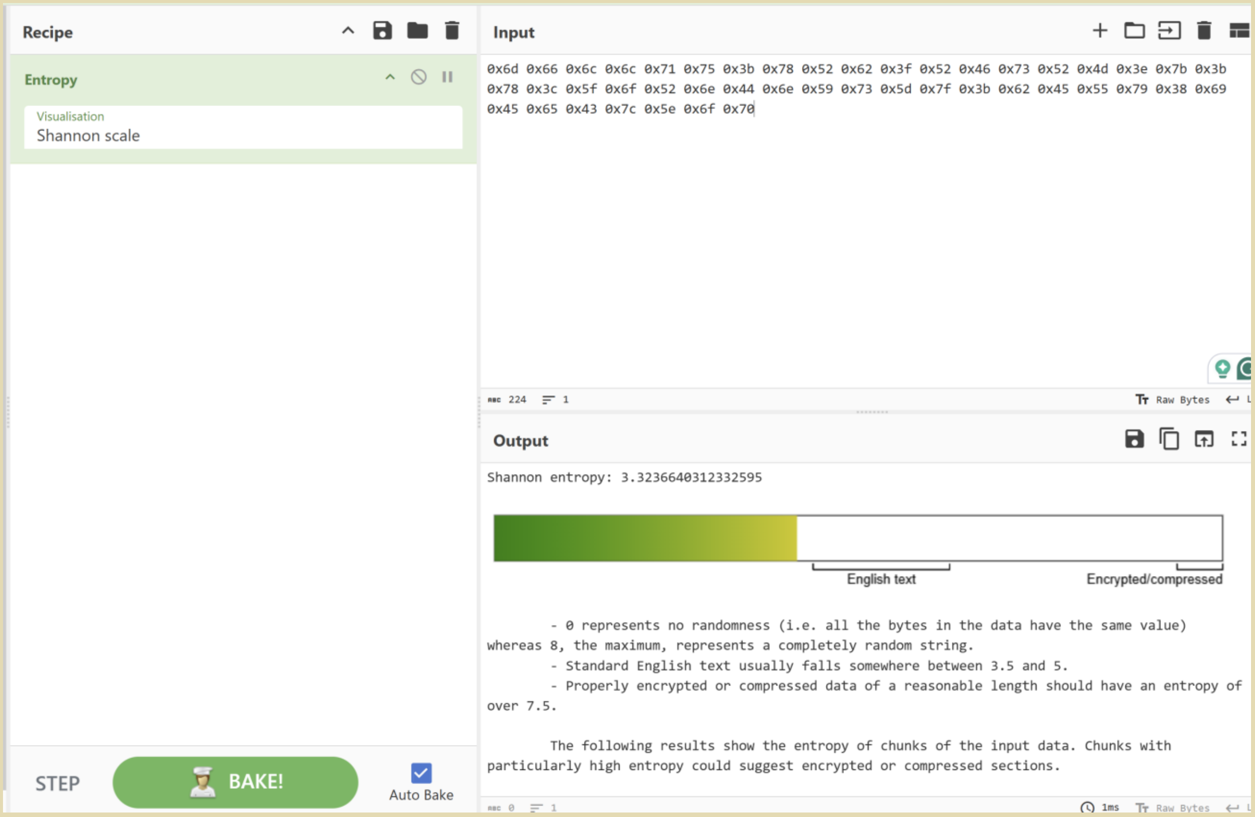
“Known Plaintext”

Are you familiar with known plaintext attacks?

0x6d 0x66 0x6c 0x6c 0x71 0x75 0x3b 0x78 0x52 0x62 0x3f
0x52 0x46 0x73 0x52 0x4d 0x3e 0x7b 0x3b 0x78 0x3c 0x5f
0x6f 0x52 0x6e 0x44 0x6e 0x59 0x73 0x5d 0x7f 0x3b 0x62
0x45 0x55 0x79 0x38 0x69 0x45 0x65 0x43 0x7c 0x5e 0x6f
0x70

Author: Grant

In known plaintext attacks, attackers have access to both the ciphertext and the inputted plaintext of some cryptographic function, and they can use this knowledge to derive the key. It appears we are given a list of hex values that most likely represent the flag encrypted in some manner. For cryptography challenges, in which I'm given some ciphertext, I find it useful to look at its Shannon entropy; this can tell us how "random" the data is. CyberChef is a great all-around tool for cryptography challenges, and it includes an entropy option:

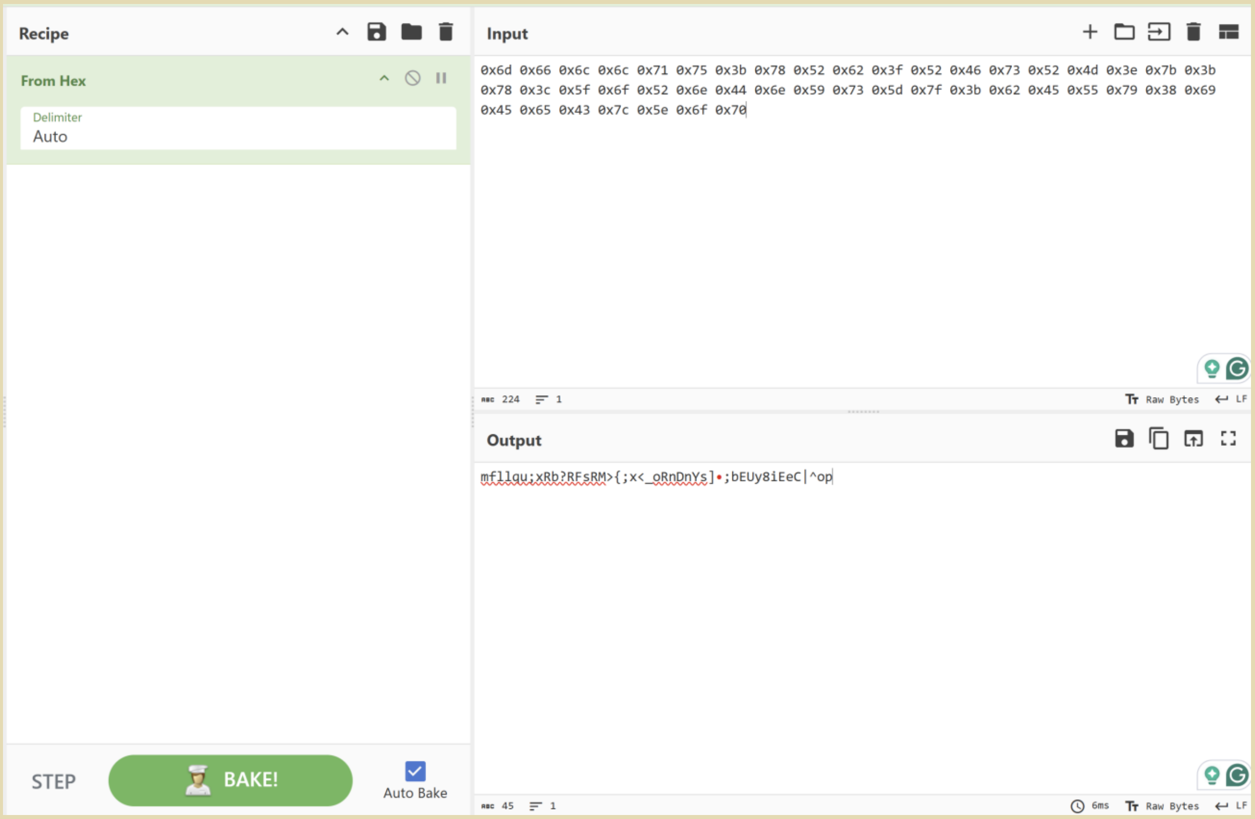


```
0x6d 0x66 0x6c 0x6c 0x71 0x75 0x3b 0x78 0x52 0x62 0x3f 0x52 0x46 0x73 0x52 0x4d 0x3e 0x7b 0x3b
0x78 0x3c 0x5f 0x6f 0x52 0x6e 0x44 0x6e 0x59 0x73 0x5d 0x7f 0x3b 0x62 0x45 0x55 0x79 0x38 0x69
0x45 0x65 0x43 0x7c 0x5e 0x6f 0x70
```

```
Shannon entropy: 3.3236640312332595
```

```
        - 0 represents no randomness (i.e. all the bytes in the data have the same value)
whereas 8, the maximum, represents a completely random string.
        - Standard English text usually falls somewhere between 3.5 and 5.
        - Properly encrypted or compressed data of a reasonable length should have an entropy of
over 7.5.

        The following results show the entropy of chunks of the input data. Chunks with
particularly high entropy could suggest encrypted or compressed sections.
```

As you can see, the entropy is close to that of normal English text; this tells me that this is very weak encryption. Popular types of ciphers that result in minimal changes in entropy include things like substitution ciphers, transposition ciphers, and xor operations. Continuing our analysis of the cipher text, let's convert these hex values (we can tell they are hex due to their 0x prefix) to ascii.


Recipe

From Hex

Delimiter: Auto

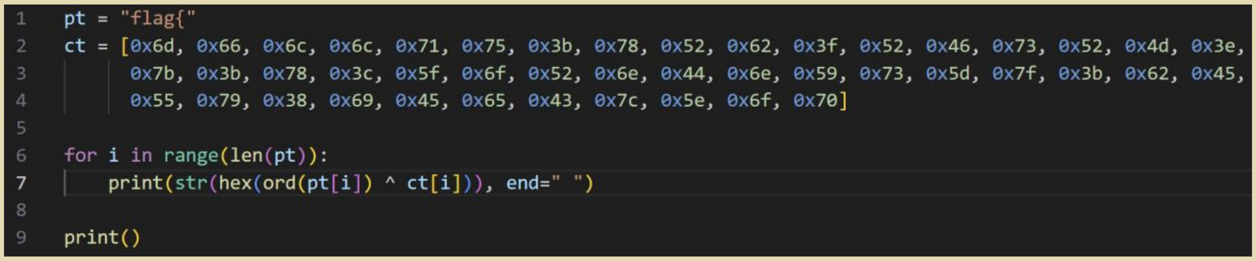
Input

```
0x6d 0x66 0x6c 0x6c 0x71 0x75 0x3b 0x78 0x52 0x62 0x3f 0x52 0x46 0x73 0x52 0x4d 0x3e 0x7b 0x3b
0x78 0x3c 0x5f 0x6f 0x52 0x6e 0x44 0x6e 0x59 0x73 0x5d 0x7f 0x3b 0x62 0x45 0x55 0x79 0x38 0x69
0x45 0x65 0x43 0x7c 0x5e 0x6f 0x70
```

Output

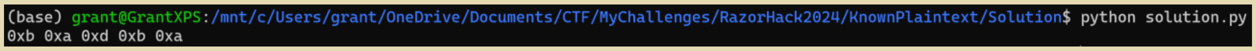
```
mfllqu;xRb?RFsRM>{;x<_oRnDnYs]•;bEUy8iEeC|^op
```

One thing to notice here is that one of the characters is not a printable character (denoted by the red circle in the output); this most likely rules out it being a substitution cipher. Let's try testing it for XOR encryption. A key property of XOR is that it is reversible; if a ^ b = c, then c ^ b = a. If we know the plaintext begins with "flag{", this means that we may be able to derive the XOR key by XORing the first 5 bytes of the cipher text with the "flag{" ascii values. To do this, we can use Python.



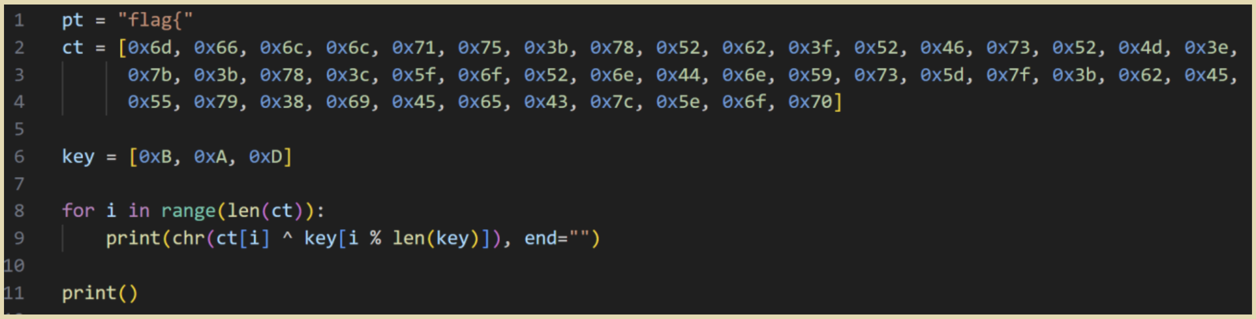
```python
pt = "flag{"
ct = [0x6d, 0x66, 0x6c, 0x6c, 0x71, 0x75, 0x3b, 0x78, 0x52, 0x62, 0x3f, 0x52, 0x46, 0x73, 0x52, 0x4d, 0x3e,
      0x7b, 0x3b, 0x78, 0x3c, 0x5f, 0x6f, 0x52, 0x6e, 0x44, 0x6e, 0x59, 0x73, 0x5d, 0x7f, 0x3b, 0x62, 0x45,
      0x55, 0x79, 0x38, 0x69, 0x45, 0x65, 0x43, 0x7c, 0x5e, 0x6f, 0x70]

for i in range(len(pt)):
    print(str(hex(ord(pt[i]) ^ ct[i])), end=" ")

print()
```

This program loops through the characters of the known plaintext (denoted by pt) and XORs their ascii values (which are obtained using the ord() function) with the corresponding bytes of the ciphertext. Running the program, we are met with the following output:



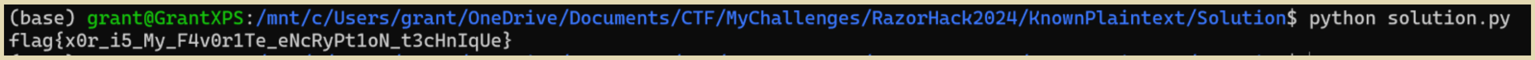
```
(base) grant@GrantXPS:/mnt/c/Users/grant/OneDrive/Documents/CTF/MyChallenges/RazorHack2024/KnownPlaintext/Solution$ python solution.py
0xb 0xa 0xd 0xb 0xa
```

Interestingly, we see the hex values 0xb, 0xa, and 0xd, before the sequence begins repeating. This indicates that this is most likely an XOR with a multi-byte key of 0xb, 0xa, 0xd. We can alter our program to use this key to decrypt the ciphertext:



```python
pt = "flag{"
ct = [0x6d, 0x66, 0x6c, 0x6c, 0x71, 0x75, 0x3b, 0x78, 0x52, 0x62, 0x3f, 0x52, 0x46, 0x73, 0x52, 0x4d, 0x3e,
      0x7b, 0x3b, 0x78, 0x3c, 0x5f, 0x6f, 0x52, 0x6e, 0x44, 0x6e, 0x59, 0x73, 0x5d, 0x7f, 0x3b, 0x62, 0x45,
      0x55, 0x79, 0x38, 0x69, 0x45, 0x65, 0x43, 0x7c, 0x5e, 0x6f, 0x70]

key = [0xB, 0xA, 0xD]

for i in range(len(ct)):
    print(chr(ct[i] ^ key[i % len(key)]), end="")

print()
```

Here, we are now XORing every byte of the ct with the multi-byte key, repeating. Running this program yields the flag:



```
(base) grant@GrantXPS:/mnt/c/Users/grant/OneDrive/Documents/CTF/MyChallenges/RazorHack2024/KnownPlaintext/Solution$ python solution.py
flag{x0r_i5_My_F4v0r1Te_eNcRyPt1oN_t3cHnIqUe}
```