Challenge Name: Temporal Shift

Challenge_Descriptions

An intercepted message from a rogue agent reveals a custom encryption scheme. It appears to be based on time and heavily relies on pseudo-random operations.

You've acquired the encryption script and a strange image that seems to hold the missing key but there's no obvious way to extract it.

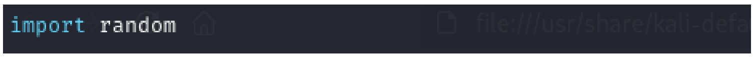
The agent used a non-standard key format and embedded it using one of his usual covert techniques. Can you uncover the key, break the cipher, and recover the flag?

File Provided:

- Ciphertext.txt
- encryptor.py
- seed.jpg

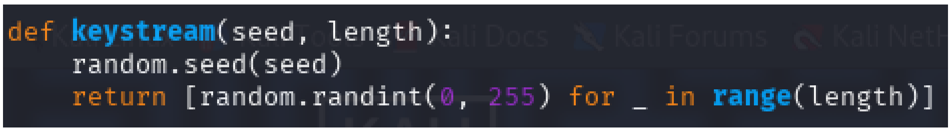
Step 1: Understand the Encryption Logic

Inspecting encryptor.py, we see:



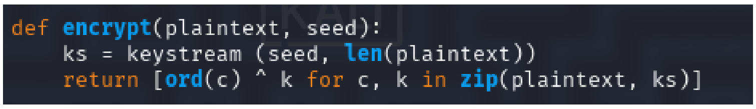
```
import random
```

- Used to generate pseudo-random numbers.



```
def keystream(seed, length):
    random.seed(seed)
    return [random.randint(0, 255) for _ in range(length)]
```

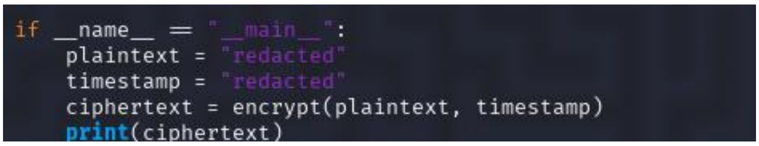
- Seeds the random number generator with seed.
- Returns a list of length random integers between 0 and 255 (inclusive).
- This acts as the **keystream** used for encryption.



```
def encrypt(plaintext, seed):
    ks = keystream (seed, len(plaintext))
    return [ord(c) ^ k for c, k in zip(plaintext, ks)]
```

- Generates a keystream the same length as the plaintext.
- XORs each character in plaintext with its corresponding number in the keystream.
- ord(c) converts each character to its ASCII value.

- Returns a list of XOR-ed numbers (the ciphertext).



```
if __name__ == "__main__":
    plaintext = "redacted"
    timestamp = "redacted"
    ciphertext = encrypt(plaintext, timestamp)
    print(ciphertext)
```

- **plaintext**: the message to encrypt (redacted in the image).
- **timestamp**: used as the **random seed** for the keystream.
- **ciphertext**: the result of XOR encryption between plaintext and generated keystream.
- The ciphertext is printed as a list of integers.

Key Concept:

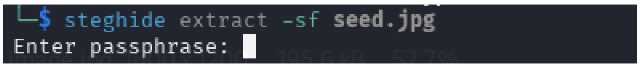
This is an implementation of a **stream cipher** using XOR encryption:

- The same seed (timestamp) must be used for **decryption**.
- If you know the timestamp and the ciphertext, you can XOR again to recover the plaintext.

Step 2: Investigate seed.jpg

Since it's a seed image, maybe it related with an experienced player may try common steganographic methods.

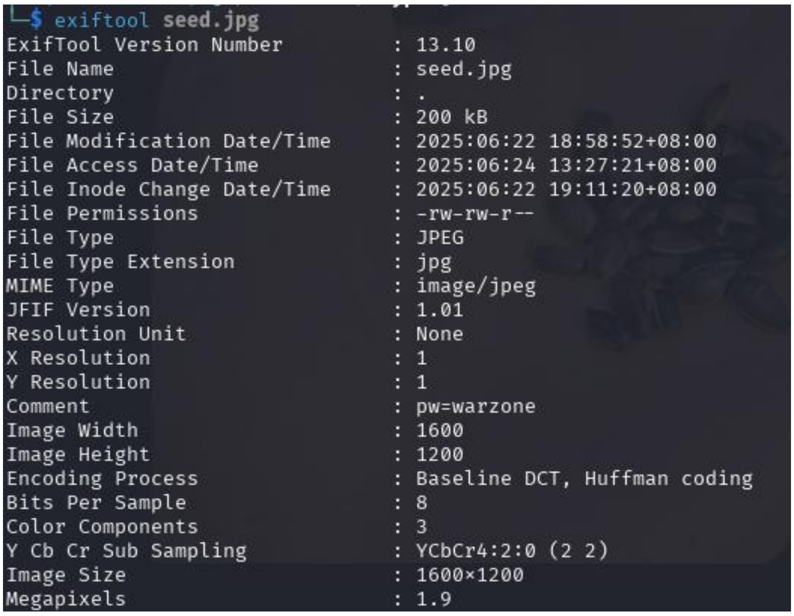
They might try:



```
└─$ steghide extract -sf seed.jpg
Enter passphrase: █
```

You can't extract the data directly from the image because it prompts for a password. You'll need to find the password first.

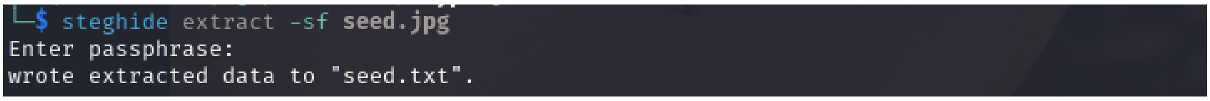
Maybe we can check the image metadata using **exiftool**



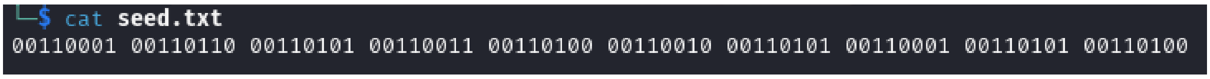
```
└─$ exiftool seed.jpg
ExifTool Version Number         : 13.10
File Name                       : seed.jpg
Directory                       : .
File Size                       : 200 kB
File Modification Date/Time     : 2025:06:22 18:58:52+08:00
File Access Date/Time           : 2025:06:24 13:27:21+08:00
File Inode Change Date/Time     : 2025:06:22 19:11:20+08:00
File Permissions                : -rw-rw-r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Resolution Unit                 : None
X Resolution                    : 1
Y Resolution                    : 1
Comment                         : pw=warzone
Image Width                     : 1600
Image Height                    : 1200
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 1600x1200
Megapixels                      : 1.9
```

We found the password at comment section **"pw=warzone"**

Step 3: Extracted Seed Data

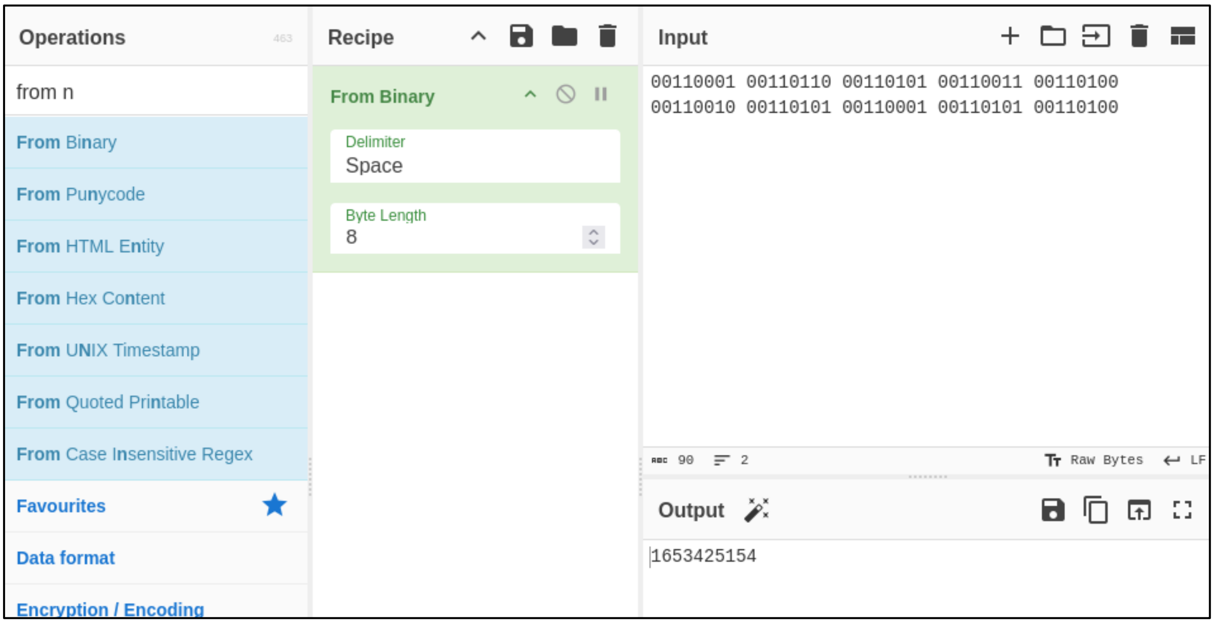


```
└─$ steghide extract -sf seed.jpg
Enter passphrase:
wrote extracted data to "seed.txt".
```



```
└─$ cat seed.txt
00110001 00110110 00110101 00110011 00110100 00110010 00110101 00110001 00110101 00110100
```

Since we obtained the seed in binary form, we need to convert it into an integer to recover the key.



From Binary

Delimiter: Space

Byte Length: 8

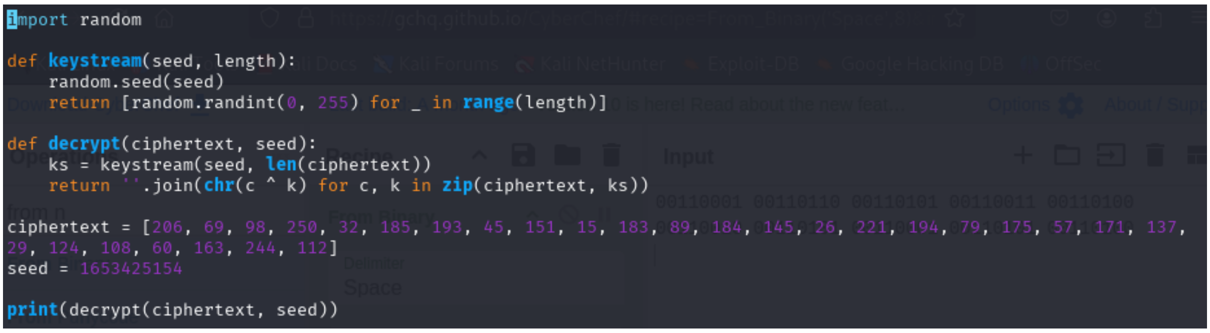
```
00110001 00110110 00110101 00110011 00110100
00110010 00110101 00110001 00110101 00110100
```

```
1653425154
```

Now we have the key (seed), **1653425154.**

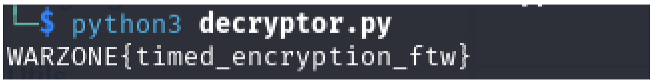
Step 4: Decrypting the Ciphertext

Now that we have the key, we can decrypt the ciphertext. To do this, we need to understand the encryptor.py file. Based on our understanding, we'll need to write a script to help us perform the decryption.



```python
import random

def keystream(seed, length):
    random.seed(seed)
    return [random.randint(0, 255) for _ in range(length)]

def decrypt(ciphertext, seed):
    ks = keystream(seed, len(ciphertext))
    return ''.join(chr(c ^ k) for c, k in zip(ciphertext, ks))

ciphertext = [206, 69, 98, 250, 32, 185, 193, 45, 151, 15, 183, 89, 184, 145, 26, 221, 194, 79, 175, 57, 171, 137,
29, 124, 108, 60, 163, 244, 112]
seed = 1653425154

print(decrypt(ciphertext, seed))
```

Final Flag:



```
└─$ python3 decryptor.py
WARZONE{timed_encryption_ftw}
```