

E.1

The flag is found on page 8 of the worksheet.

E.2

Based off the named of the challenge, I assumed a Caesar cipher aka substitution cipher was used on the plaintext to obtain the ciphertext. I obtained the flag by piping the encrypted text through the rot13 utility.

```
(kali@kali)-[~]  
$ echo PF2107{Gurer_V5_n_YbG_Bs_Uby3f_Va_Gu1f_Ba3} | rot13  
CS2107{There_I5_a_LoT_Of_Ho13s_In_Th1s_On3}
```

E.3

This challenge requires us to find the image among a file of images that matches the MD5 sum given. To find the file, I used the md5sum utility in linux to compute the MD5 of all the images in the folder, then piped this output to the grep (search) function with the MD5 hash I am looking for. The output is the filename I was searching for.

```
(kali@kali)-[~]  
$ md5sum /home/kali/Desktop/dist/* | grep 8ac8fb3045e78a65a9df5685fe715dff  
8ac8fb3045e78a65a9df5685fe715dff /home/kali/Desktop/dist/y4_f0und_m3_am1g0s_07331.png
```

Using this filename, I used the sha1sum utility to compute the SHA1 hash of the file.

```
(kali@kali)-[~]  
$ sha1sum /home/kali/Desktop/dist/y4_f0und_m3_am1g0s_07331.png  
503e164c707c5d77f17c7a51b3a1e9860694a118 /home/kali/Desktop/dist/y4_f0und_m3_am1g0s_07331.png
```

Concatenating the resultant filename and SHA1 hash of the file, I obtained the flag:
CS2107{y4_f0und_m3_am1g0s_07331_503e164c707c5d77f17c7a51b3a1e9860694a118}

M1.

From the question, we can infer that a substitution cipher was used on the ciphertext, because of the phrase “my keycaps got switched around”. All that is needed to do is to find the substitution cipher key to decrypt the ciphertext. I used a substitution cipher decoder on planetcalc.com, input the ciphertext, and obtained the plaintext. The key used to encrypt and decrypt the ciphertext was also derived.

Substitution cipher breaker

Encrypted text
 nxkfzkgma pgpvarbr dc okmcok p ykrripik.

Bg mcgmzvrbcg, nxkfzkgma pgpvarbr br p dkmjgbfzk djpd jpr lkkg zrko ncx mkgdzxbkr dc pgpvaqk pgo okmcok kgmxatdko ykrripik.
 Bd br lprko cg djg txbgmbtvk djpd mxxdpbg vkddkxr cx ixcztr cn vkddkxr pttkpx ycxk nxkfzkgdva bg p ibekg tbkmk cn dkwd, pgo la
 pgpvaqbgj djg nxkfzkgma cn djkrk vkddkxr, bd br terrblvk dc ipbg bgrbjid bgdc djg zgokxvabgi rdzmdzkk cn djg kgmcoko ykrripik.
 Sihvk nxkfzkgma nnnvarbr inc lkkn unvikva rztkvknko la edikv ukdient hnd xkvnhr n zrkzvu dees ha mkvzdoha mhvmzurdnmkr.

Key to decrypt the message
 YIOTVQNKGHEBCFDAZSWPJLXRMU

Key used to encrypt the message
 PLMOKNIJBHUVYGCTFXRDZESWAQ

Decrypted text
 FREQUENCY ANALYSIS IS A TECHNIQUE THAT HAS BEEN USED FOR CENTURIES TO ANALYZE AND DECODE ENCRYPTED MESSAGES. THE PRINCIPLE BEHIND THIS METHOD IS BASED ON THE FACT THAT CERTAIN LETTERS OR GROUPS OF LETTERS APPEAR MORE FREQUENTLY IN A GIVEN PIECE OF TEXT. BY ANALYZING THE FREQUENCY OF THESE LETTERS OR GROUPS OF LETTERS, IT IS POSSIBLE TO GAIN INSIGHT INTO THE UNDERLYING STRUCTURE OF THE ENCODED MESSAGE, AND POTENTIALLY EVEN DECRYPT IT.

The instructions on how to format the flag can be found in the 4th paragraph of the plaintext. Following the instructions, I obtained the flag:

CS2107{Frequency_analysis_is_much_easier_on_longer_text}

M2.

We are given the information that the file was (1) encrypted with AES ECB, and that (2) the header was removed before encryption.

1. We already know that AES ECB is not secure because each 128 bit block of plaintext is encrypted with the same key and that it is deterministic. Hence we don't even need to decrypt the file to uncover information that the encrypted image leaks.
2. The file given in the question is not readable by an image viewer since its header was removed, but we observe that the image file had the extension .ppm before it was encrypted. We are also given the unique piece of information that the image was 940 pixels wide. We can simply input our own header from all the information we can derive to make the encrypted file readable again!

I first derived the file size.

```
(kali@kali)-[~]
$ ls -l /home/kali/Desktop/body.ppm.ecb
-rw-r--r-- 1 kali kali 2549297 Feb 19 04:22 /home/kali/Desktop/body.ppm.ecb
```

Knowing that the filesize is 2549297 bits, and that PPM uses 24 bits per pixel, we can calculate what is the supposed height of the image.

$$2549297 * 8 = 20394376$$

$$20394376 / 24 = 849765.6667$$

$$849765.6667 / 940 = 904$$

The image is 904 pixels tall!

A quick google search returned the valid format of ppm file headers.

The following are all valid PPM headers.

Header example 1

```
P6 1024 788 255
```

Header example 2

```
P6
1024 788
# A comment
255
```

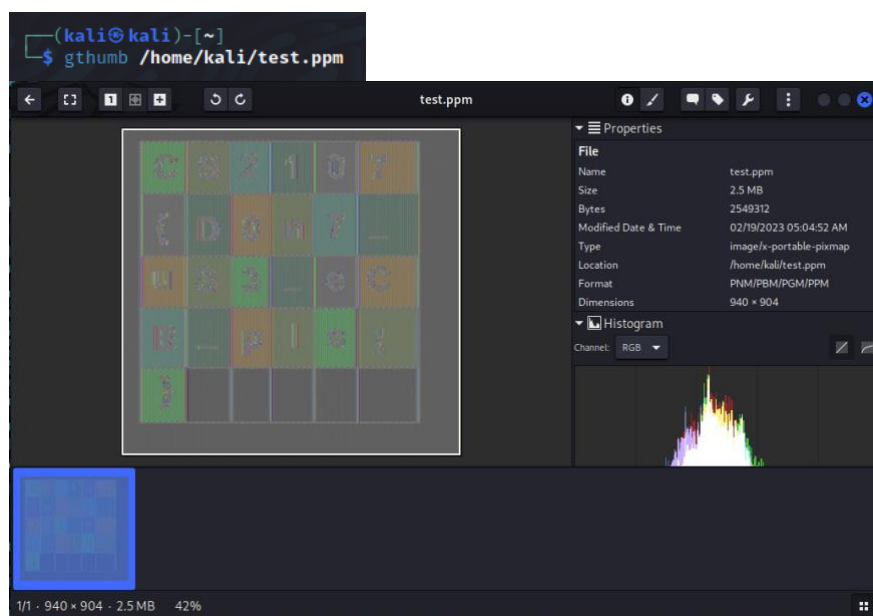
Now knowing that the image is 940 pixels in width and 904 pixels in height, we can now go on to use that information to add a header to the encrypted image.

```
GNU nano 6.4 header.txt
P6
940 904
255
```

I then concatenated the header to the encrypted image.

```
(kali㉿kali)-[~]
$ cat header.txt /home/kali/Desktop/body.ppm.ecb > test.ppm
```

Using the gthumb image viewer to view the combined file, I easily uncovered the message from the encrypted image.



CS2107{D0n7_uS3_eCB_pls!}

M3.

As the name of the challenge implies, this challenge requires us to take advantage of the fact that CRC32 is not cryptographically secure, since it can be easily manipulated. CRC32 takes an arbitrarily large input space and hashes it to a fixed 32-bit size value. Due to the nature of hashes, collisions can occur when several different inputs hash to the same value. Doing a quick calculation to calculate the probability of collisions using the formula $1 - e^{(-N^2/2k)}$, where N is the possible number of input values (in this case 100×10^9 since the string could have 100×10^9 possibilities "Hehe, my value is X", and X ranges from 1 to 100×10^9) and k is the output space of CRC32), we see that there is a 91.6% probability of collision! This makes reverse engineering the correct input hash possible – which is what the input value is looking for.

Of course, it is also possible to naively pre-compute the CRC32 hash of all possibilities of "Hehe, my value is X" with X ranging from 1 to 100×10^9 , and then do a look-up of the hash given on the server platform prompt. But because N is large, it is computationally expensive to do so. Hence, I did not take this approach but did the reverse-engineering instead.

To tackle this question, I made use of an open source CRC32 reverse engineering tool: <https://github.com/skysider/crc32hack>

Simply using this tool to reverse engineer the output gave many possible input strings that hashed to the same value.

```
jowayn@nvr8 ~ % python crc32.py reverse 0x9603a0b6
4 bytes: {0xc2, 0x22, 0x36, 0xe9}
verification checksum: 0x9603a0b6 (OK)

alternative 5 bytes:
alternative: 9GE1d (OK)

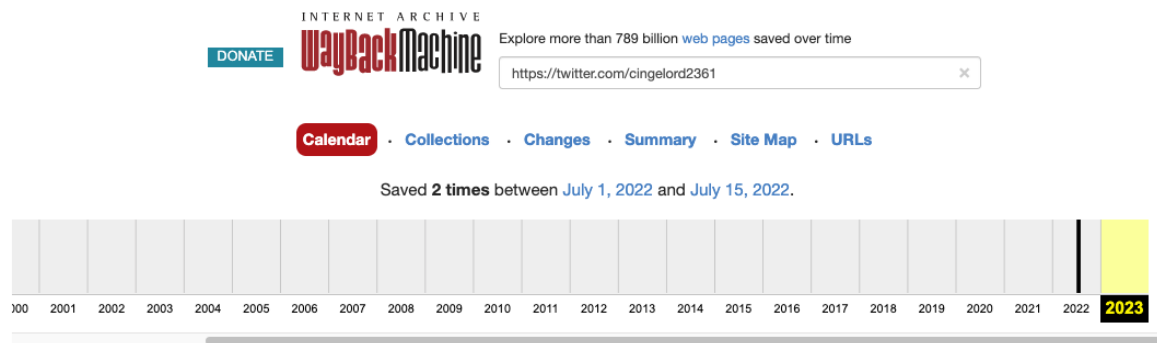
alternative 6 bytes:
alternative: 00b6pQ (OK)
alternative: 4DCftb (OK)
alternative: 7dH57r (OK)
alternative: GDv1ff (OK)
alternative: GX90gr (OK)
alternative: KV2Rj9 (OK)
alternative: NRnbpC (OK)
alternative: TIYQ8y (OK)
alternative: X7nB0b (OK)
alternative: XGR352 (OK)
alternative: _cDAwA (OK)
alternative: fWK8Gn (OK)
alternative: iDZeam (OK)
alternative: j53wNa (OK)
alternative: k5rFUx (OK)
alternative: oAS6QK (OK)
alternative: qchHpJ (OK)
alternative: tg4xj0 (OK)
alternative: v7BK7i (OK)
alternative: vzovZa (OK)
alternative: xupFfo (OK)
(base) jowayn@nvr8 ~ %
```

Inputting the first string given granted me the flag.

```
((base) jowayn@nvr8 ~ % nc cs2107-ctfd-i.comp.nus.edu 10520
I am going to generate a number between 1 and 1000000000000
I will not help you. I can only give this: 0x9603a0b6
Can you achieve happiness >9GE1d
Congratulations, you have attained nirvana: CS2107{m1nd_th3_coll15i0n}
```

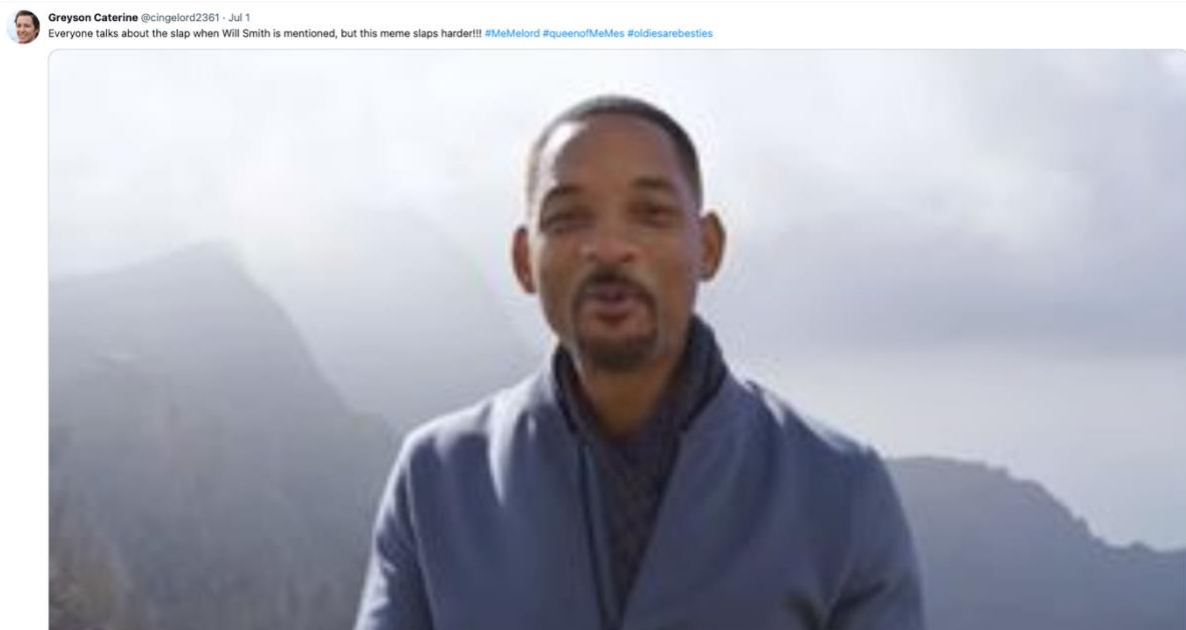
H1.

This challenge involves retrieving the password to unlock a zipped file. While most of the information was easily obtainable from Greyson Catherine's social media webpages, the only piece of information that was more challenging to obtain was her favourite movie, as it was not found on her twitter page or her CV website. However, I found out that one of the more common strategies in tackling OSINT-type challenges in CTF was to hide the information by changing the year or in backdated information. Hence, I used the following tool to try to obtain saved pages of Greyson Catherine's twitter profile.



As it turns out, there were 2 archived pages of her twitter page on the following 2 occasions, each one uncovering a tweet that was since deleted.

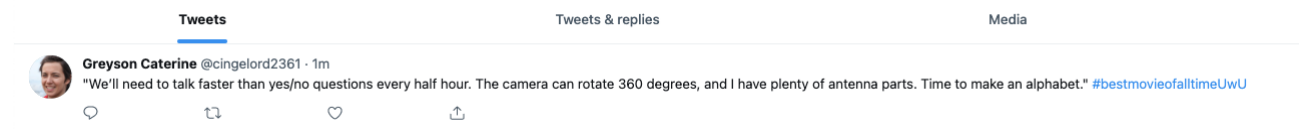
July 1st 2022:



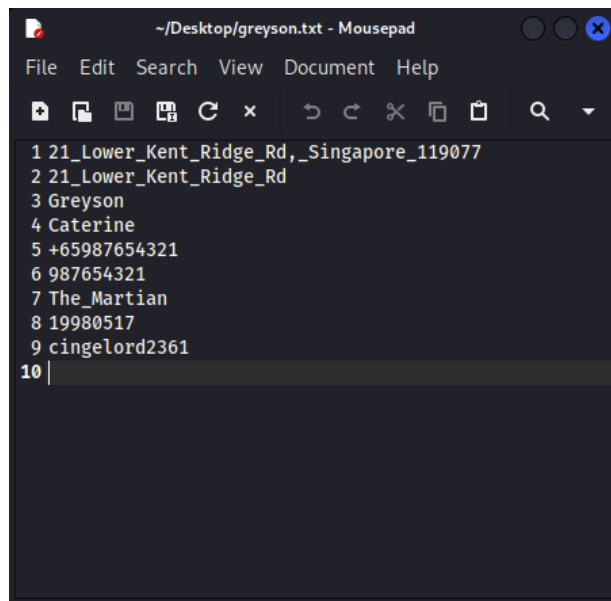
July

15th

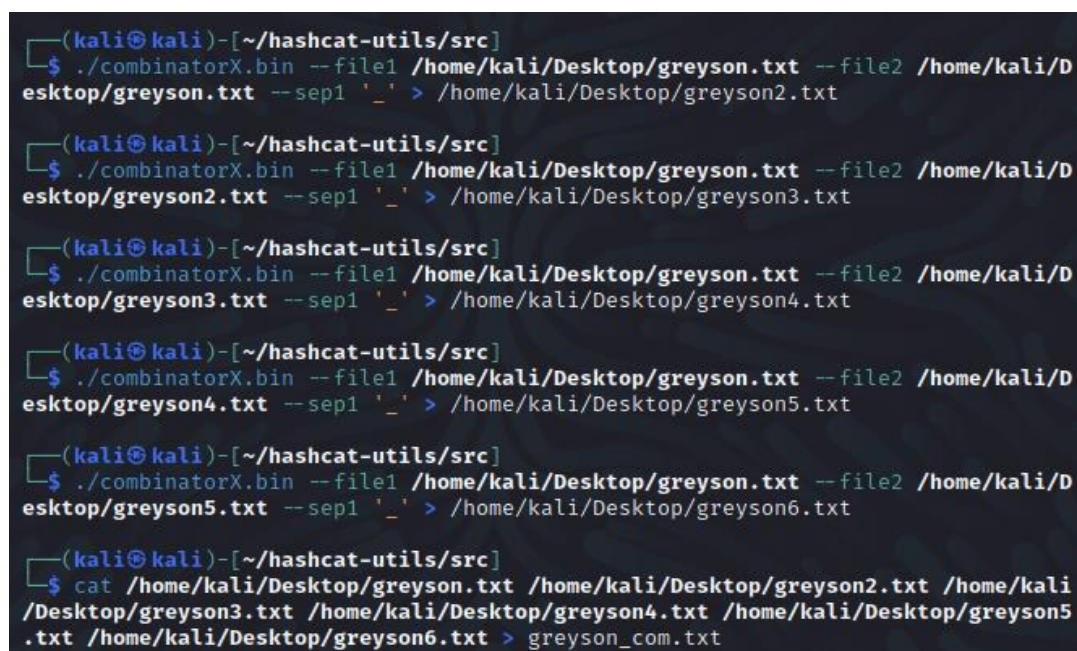
2022:



By simply searching up the quote, I deduced that her favourite movie was “The Martian”. I then created the following text file with the information that was most likely going to appear in her password.



Given the information that her password was made up of 1 or more of the above, I used a combinator tool in *hashcat-utils* to permutate different combinations of the above worlist, separated by an underscore. I obtained permutations of the wordlist with up to 5 words and then concatenated the text files into greyson_com.txt.



Equipped with a list of passwords to crack open the zip file, I made use of John The Ripper to find a hash of the passwords in the wordlist that matches that of the hash of the correct password. First, I used the *zip2john* tool to obtain the password hash of the zip file.

```
(kali@kali)-[~/Desktop]
$ zip2john /home/kali/Desktop/flag.zip > zip.txt
```

Using John The Ripper to find a matching hash with that of the zip file against my wordlist, the resulting output returned the password!

```
(kali@kali)-[~/Desktop]
$ john --wordlist=greyson_com.txt /home/kali/Desktop/zip.txt
Using default input encoding: UTF-8
Loaded 1 password hash (ZIP, WinZip [PBKDF2-SHA1 256/256 AVX2 8x])
Cost 1 (HMAC size) is 53 for all loaded hashes
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
Greyson_Caterine_cingelord2361_The_Martian (?)
1g 0:00:00:00 DONE (2023-02-22 06:24) 7.692g/s 63015p/s 63015c/s 63015C/s 21_Lowe
r_Kent_Ridge_Rd,_Singapore_119077..21_Lower_Kent_Ridge_Rd,_Singapore_119077_+6598
7654321_987654321_The_Martian_Caterine
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

Unlocking the zip file with the password returned the flag,
C2107{th15_i5_h0w_w3_g3t_att4ck3d_On_th3_int3rn3t}

H2.

Interacting with the challenge on the challenge server, my first observation was that the encryption key must have changed with each instance I interacted with the challenge server, as it produced a different set of ciphertexts each time. For the sake of analysis, I will be using this set of ciphertexts.

```
(kali@kali)-[~]
$ nc cs2107-ctfd-i.comp.nus.edu.sg 5001
Ciphertext0 in hex: 8f12839b00315646db7d1640099564c94b8e2ead
Ciphertext1 in hex: 8f12839b017a54b717fba649efce1c19616fa3bf
Ciphertext2 in hex: 8f12839b9bafa8546ac09ce393ec811d75301b9f
```

In this set of ciphertexts, I observed that the first 4 leading bytes were identical. This observation was key to defeating this challenge, as this proves to be a serious vulnerability in the encryption algorithm. Not only does this suggests some form of padding of the plaintext before encryption, it also tells us that this encryption doesn't have the desirable property of an "avalanche" effect, because changes in the plaintext do not affect the entire ciphertext.

In this question's implementation of fast-encrypt, a 128-bit key is randomly generated. Then t is obtained by using the master key to encrypt 128 bits of 0s.

The sessionkey is then obtained by extracting the leading 24 bits of t. This acts as a temporary key derived from the master key, which changes from each interaction with the server.

The plaintexts are then encrypted using the sessionkey. The crucial vulnerability to this algorithm lies here, in that encrypting the plaintext, 32 bits of 1s (4 bytes of xFF) are prepended to the plaintext!

```
pt = b'\xFF' * 4 + pt]
```

This allows us to pre-compute a “rainbow table” of encrypting 32 bits of 1s using all possible 24-bit sessionkeys. This also explains the observation made above about the same 4 leading bytes. Given a ciphertext, we just need to find the first 32 bits of the ciphertext that matches with what we have in our look-up table, and we can find out what was the sessionkey used! This is good enough to be able to decrypt all the ciphertexts because of the design of the encrypt function used. In this design, sessionkey is used as the parameter into random.seed(), and since we know this function is deterministic, we are easily able to restore the random number generated by the encrypt function in order to decrypt the ciphertext through an XOR of the ciphertext with the number generated to obtain the plaintext.

Using the same encrypt function as in the server.py file, I computed the rainbow table with the below code. I iterated across $[0, 2^{24}]$ possible session keys to use as the session_key in encrypting 4 bytes of 1s.

```
import pandas as pd
import random

def xor(a, b):
    return bytes((i ^ j) for i, j in zip(a, b))

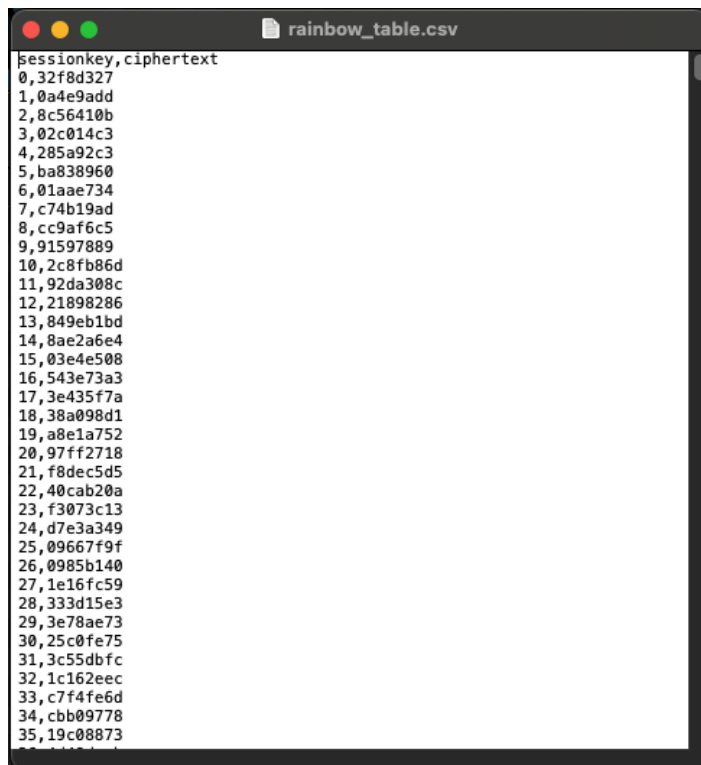
def encrypt(allones, sessionKey):
    random.seed(sessionKey)
    return xor(random.randbytes(20), allones)

# Initialize dataframe
df = pd.DataFrame(columns=["sessionkey", "ciphertext"])

# Iterate through all possible session keys
for i in range(2**24):
    session_key = i
    ciphertext = encrypt(b'\xFF'*4, session_key) #Generate ciphertext
    df = df.append({"sessionkey": session_key, "ciphertext": ciphertext.hex()}, ignore_index=True)
    print(i)

# Save the DataFrame to csv
df.to_csv("rainbow_table.csv", index=False)
```


The following rainbow table was obtained. The left column indicates the session_key used to encrypt 4 bytes of 1s, and the right column indicates the encrypted 4 bytes using the corresponding session_key in that row.



The image shows a screenshot of a text editor window titled "rainbow_table.csv". The window contains a list of 36 rows, each representing a session key and its corresponding ciphertext. The first row is a header: "sessionkey,ciphertext". The subsequent rows are numbered from 0 to 35, with each row containing a session key followed by a comma and its ciphertext. The session keys are hexadecimal strings, and the ciphertexts are also hexadecimal strings.

sessionkey	ciphertext
0	32f8d327
1	0a4e9add
2	8c56410b
3	02c014c3
4	285a92c3
5	ba838960
6	01aae734
7	c74b19ad
8	cc9af6c5
9	91597889
10	2c8fb86d
11	92da308c
12	21898286
13	849eb1bd
14	8ae2a6e4
15	03e4e508
16	543e73a3
17	3e435f7a
18	38a098d1
19	a8e1a752
20	97ff2718
21	f8dec5d5
22	40cab20a
23	f3073c13
24	d7e3a349
25	09667f9f
26	0985b140
27	1e16fc59
28	333d15e3
29	3e78ae73
30	25c0fe75
31	3c55dbfc
32	1c162eec
33	c7f4fe6d
34	cbb09778
35	19c08873

Next, I went on to write code that parses the 3 lines of ciphertexts in a text file, then looks up the rainbow table to find a match of the first 4 bytes of the first ciphertext with the encrypted 4 bytes of 1s in the rainbow table. After a match is found, the particular session_key corresponding to the one used to encrypt the original plaintext is then used to decrypt the 3 ciphertexts. The output is the original 3 plaintexts that were encrypted!

```
from Crypto.Util.number import long_to_bytes
import random
import os
import csv

with open('ciphertexts.txt', 'r') as file:
    lines = file.readlines()

cipher0 = lines[0].split(": ")[1]
cipher1 = lines[1].split(": ")[1]
cipher2 = lines[2].split(": ")[1]

# Define the xor function
def xor(a, b):
    return bytes(i ^ j for i, j in zip(a, b))

# Define the decrypt function
def decrypt(ct, sessionKey):
    random.seed(sessionKey)
    return xor(random.randbytes(20), ct)[4:]

def lookup(cipher0, cipher1, cipher2):
    n = 3
    c0 = bytes.fromhex(cipher0)
    c1 = bytes.fromhex(cipher1)
    c2 = bytes.fromhex(cipher2)
    ciphertexts = [c0, c1, c2]
    prefix = cipher0[:8]
    print(f"First 4 bytes: {prefix}")
    with open("rainbow_table.csv", newline='') as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            if row["ciphertext"].startswith(prefix):
                print("Match Found")
                sessionKey = int(row["sessionkey"])
                print(f"sessionkey used: {sessionKey}")

                # Decrypt each ciphertext using the session key
                plaintexts = [decrypt(ct, sessionKey) for ct in ciphertexts]

                # Print the plaintexts
                for i in range(n):
                    my_bytes = plaintexts[i]
                    hex_str = ''.join(['{:02x}'.format(b) for b in my_bytes])
                    print(hex_str)
                    break
            else:
                print("No match found")

# call the lookup function with the extracted ciphertexts
lookup(cipher0, cipher1, cipher2)
```

The output of the above lookup and decryption code is as follows.

```
===== RESTART: /home/kali/Desktop/fast_decrypt.py ===
First 4 bytes: 8f12839b
Match Found
sessionkey used: 1368353
840f132b09b92a2b3605e320346993a4
854411dac53f9a22d05e9bf01e881eb6
1f91ed39b804a088ac7c06f40ad7a696
```

Entering the corresponding plaintexts, the server returned the flag
CS2107{H4sH_tABl3s_4rE_tH3_GreA7E5t_TaB13s}

```
(kali㉿kali)-[~]  
$ nc cs2107-ctfd-i.comp.nus.edu.sg 5001  
Ciphertext0 in hex: 8f12839b00315646db7d1640099564c94b8e2ead  
Ciphertext1 in hex: 8f12839b017a54b717fba649efce1c19616fa3bf  
Ciphertext2 in hex: 8f12839b9bafa8546ac09ce393ec811d75301b9f  
  
Please enter Plaintext0 in hex without 0x at the start: 840f132b09b92a2b3605e3203  
46993a4  
  
Please enter Plaintext1 in hex without 0x at the start: 854411dac53f9a22d05e9bf01  
e881eb6  
  
Please enter Plaintext2 in hex without 0x at the start: 1f91ed39b804a088ac7c06f40  
ad7a696  
Congrats! Here is your flag!  
CS2107{H4sH_tABl3s_4rE_tH3_GreA7E5t_TaB13s}
```