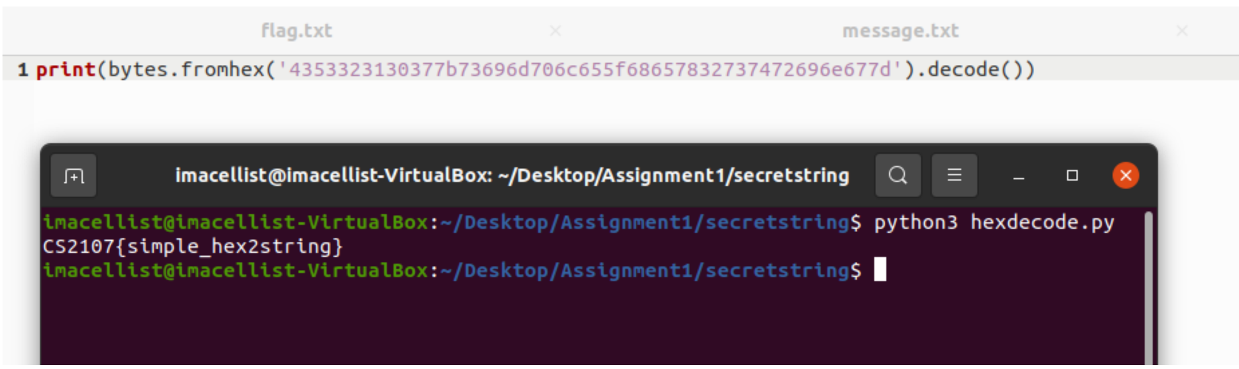
Section A: Warmup

A.1 Secret String (5 Points)

The message was found near the end of the assignment pdf but it was in hexadecimal.

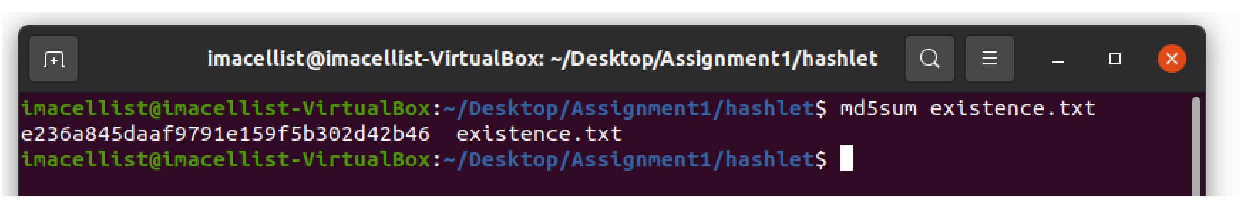
> To reward you for reading this far, this might be helpful for the first challenge:
> `4353323130377b73696d706c655f686578327374472696e677d`

We can use an online hexadecimal to ascii converter or just decode it using python to get the flag CS2107{simple_hex2string}.



A.2 Hashlet (5 Points)

We just need to find the MD5 checksum of existence.txt and submit the flag as CS2107{e236a845daaf9791e159f5b302d42b46}.



A.3 Hashmap (7 Points)

Since the hashes are hexadecimal numbers of length 40, the hash function must be SHA-1 which produces a 160 bit digest.

Luckily, the hash for Password1 can be decrypted with the help of online tools to get P@ssw0rd1!.

Decrypt Hash Results for: 073158933d0377d419cd1e5dfcb4eafde8d1dd8a

Algorithm	Hash	Decrypted
sha1	073158933d0377d419cd1e5dfcb4eafde8d1dd8a	P@ssw0rd1!

Not so lucky for the Flag though.

No hashes found for 9d9eea545804f3a4edf7315c5325a4e55268420d

However, since we know that Password2 is in the range [AAAAAA-ZZZZZZ] or 6 Uppercase Letters, we can brute force with every possible value of Password2 and check if the hash value matches the hash of the Flag.

```python
from hashlib import sha1

uppercase = ['A', 'B', 'C','D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

print('073158933d0377d419cd1e5dfcb4eafde8d1dd8a' == sha1(b'P@ssw0rd1!').hexdigest())

for i in range(26):
	for j in range(26):
		for k in range(26):
			for l in range(26):
				for m in range(26):
					for n in range(26):
						key = bytes(uppercase[i] + uppercase[j] + uppercase[k] + uppercase[l] + uppercase[m] + uppercase[n], 'ascii')
						hashval = sha1(b'CS2107{P@ssw0rd1!_' + key + b'}').hexdigest()
						#print(hashval)
						if hashval == '9d9eea545804f3a4edf7315c5325a4e55268420d':
							print("FOUND!")
							print(key)
							print(b'CS2107{P@ssw0rd1!_' + key + b'}')

print("done")
```

```
imacellist@imacellist-VirtualBox: ~/Desktop/Assignment1/hashmap
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/hashmap$ python3 hash.py
True
FOUND!
b'BRUTED'
b'CS2107{P@ssw0rd1!_BRUTED}'
```

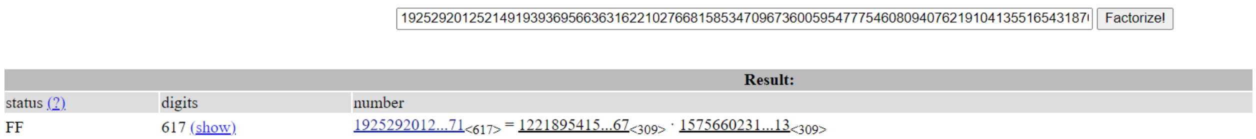
We find that CS2107{P@ssw0rd1!_BRUTED} gives the right hash value.

Section B: Main

B.1 Elementary RSA (7 Points)

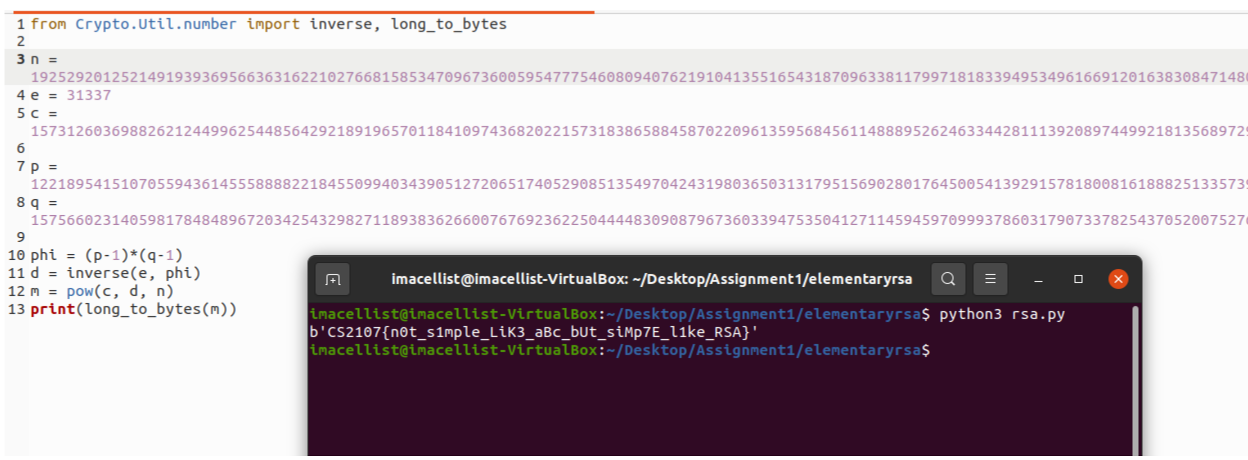
We are given n, e and c.

The problem of getting the RSA private key from the public key is as difficult as the problem of factoring n. Fortunately, we are able to factorise the n given to us using factordb.com, which is a database that stores known factorizations of numbers.



Now that we have the 2 random primes p and q, we can determine the euler totient function as phi = (p-1)*(q-1). d can then be determined as the multiplicative inverse of e modulo phi. We now have the private key (d, n) which can be used to decrypt ciphertext c by computing c^d(mod n).

We get the message m to be CS2107{n0t_s1mple_LiK3_aBc_bUt_siMp7E_l1ke_RSA}.



B.2 Secret XOR Service (10 Points)

We are given the source code XOR.py. A random 32 byte key is generated using os.urandom(32). However, the way the key is extended to match the length of the plaintext can be exploited.

extend_key(key, size) multiplies the original random key by size//len(key) and then matches the exact length of the plaintext by adding the slice of key[:size%len(key)].

As the flag is appended to our input and then encrypted, we can determine the length of the flag by connecting to the service and passing in nothing. We find the length of the flag is 27 bytes.

```
===== MY SECRET XOR SERVICE =====

Enter your favourite phrase (in hex):
Here is your encrypted result (in hex): b8e613506f8c6a84a0e1f4d87b3665f6fb4ea36fcbd8d7a7b53742
```

Since we know (x XOR 0) = x, we can pad the flag with 32 bytes of 0 to return the entire key before the encrypted flag.

```
===== MY SECRET XOR SERVICE =====

Enter your favourite phrase (in hex):  000000000000000000000000000000000000000000000000000000000000
0000
Here is your encrypted result (in hex): 83d91269208a2155492adc43af5e84ec1f2da0f05e7fa5553d5017e636e
84532c08a205810bd5a383075a473dd01b2dc2872c3826a1cce66592d1d
```

Using this ciphertext, we get the first 27 bytes of the key with ciphertext[:54] and XOR with the encrypted flag ciphertext[64:] to get the original 27 byte flag since (C XOR K) = (P XOR K XOR K) = P.

Flag found is CS2107{my_x0r_607_cr4ck3d}.

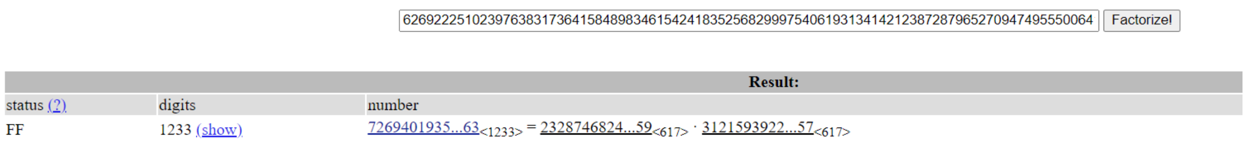
```python
def xor(a, b):
    return bytes([i^j for i,j in zip(a,b)])

def extend_key(key, size):
    return key*(size//len(key)) + key[:size%len(key)]

ciphertext = "83d91269208a2155492adc43af5e84ec1f2da0f05e7fa5553d5017e636e84532c08a205810bd5a383075a473dd01b2dc2872c3826a1cce66592d1d"
key = ciphertext[:54]
flag = ciphertext[64:]
hexflag = xor(bytes.fromhex(flag), bytes.fromhex(key)).hex()


print(bytes.fromhex(hexflag).decode())
```

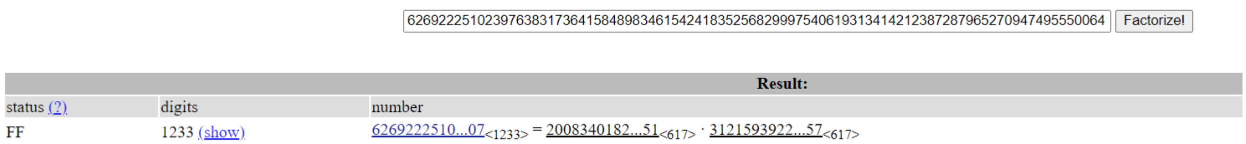
```
imacellist@imacellist-VirtualBox: ~/Desktop/Assignment1/secretxorservice
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/secretxorservice$ python3 solve.py
CS2107{my_x0r_607_cr4ck3d}

imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/secretxorservice$
```

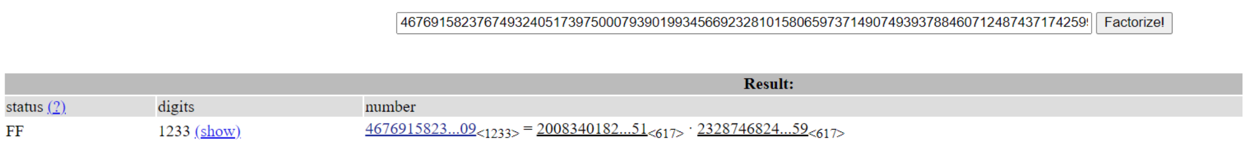
B.3 Secondary RSA (12 Points)

The flag is encrypted 3 times by looping through ns = [q * r, p * q, p * r] and using e = 65537 and each n as the public key.

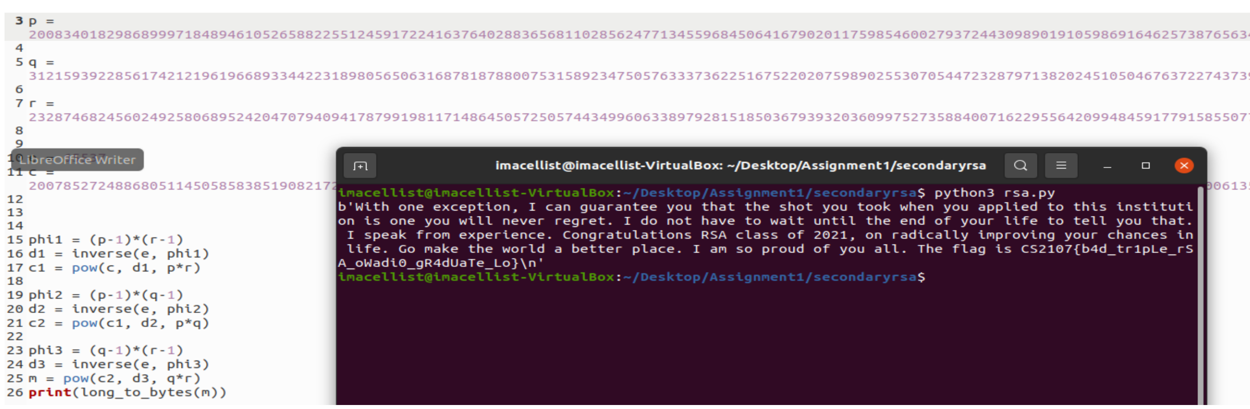
Since we are given the values of the ns array in secret.txt, maybe we can use factordb again to determine p, q and r. Fortunately, it works and we can match the repeated values to p, q and r. Alternatively, to get each prime, we could multiply 2 of the numbers and divide by the third to get the square of each prime. (e.g ns[0]*ns[1]//ns[2] = q^2*r*p//p*r = q^2). Then we can use factordb to find the roots.







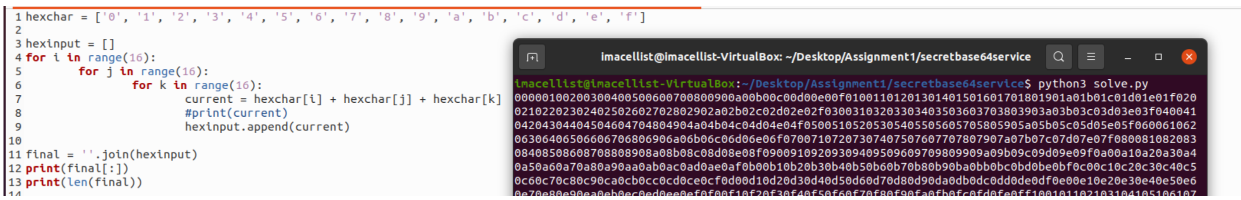
Now that we have p, q and r, we can decrypt the ciphertext 3 times in reverse order => p*r, p*q then q*r. Following the same steps in Elementary RSA to determine phi, d and the intermediate ciphertexts, we finally get a long message with the flag CS2107{b4d_tr1pLe_rSA_oWadi0_gR4dUaTe_Lo} at the end.



B.4 Secret Base64 Service (12 Points)

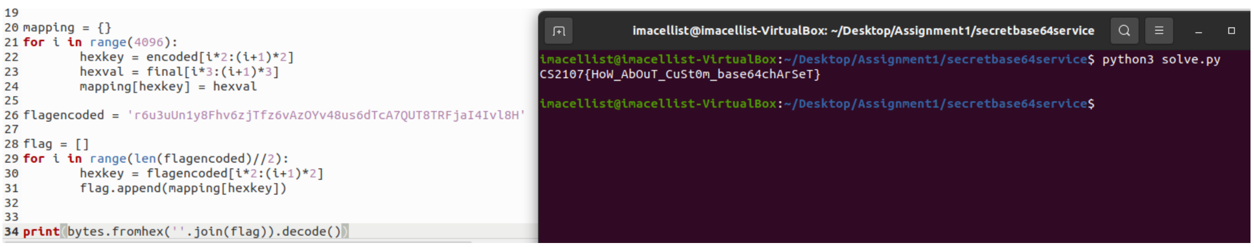
Notice that 3 hex characters (3x4 = 12 bits) map to 2 base64 characters (2x6 = 12 bits). For example, 0x000 => nn, 0x001 => nj, ...etc.

We can then generate all possible hexadecimal values with 3 hex characters
= 16x16x16 = 4098 possible combinations.



Unfortunately, Secret Base64 Service 1 does not accept the long hex input generated above. However, splitting it into 4 parts seems to work.

Using the base64 mapping for each combination of 3 hex characters to create a dictionary and getting the encoded flag from Secret Base64 Service 2, we can map the encoded flag to get the secret flag CS2107{HoW_AbOuT_CuSt0m_base64chArSeT}.



B.5 AES Good AES Me (12 Points)

From scheme.png, we know the encryption is AES-CBC with flag[1] as the IV and block size of 16 bytes. From secret.txt, we are given a partial 16 byte key, full 32 byte plaintext (P0 and P1), partial 16 byte ciphertext from the first block (C0) and full 16 byte ciphertext from the second block (C1).

To determine the 2 unknown bytes of the key, we can generate all possible ascii values and use this potential key to decrypt C1 with P1 as the IV. Since C1 = Ek(P1 XOR C0) => C0 = Dk(C1) XOR P1, using AES-CBC to decrypt C1 with P1 as the IV returns C0’. Compared with the partial C0 we are given, if the corresponding parts of C0’ match with C0, the current key we are trying must be the one. We can even determine the full C0.

```python
from Crypto.Cipher import AES

key1 = '6F738g9Zz'
key2 = 'S3j4g'

message = b'I have been blocked everywhereee'

ciphertext1 = '0df67cc18cdfc7f7605596e159d1102d'

for i in range(128):
        for j in range(128):
                key = bytes(key1 + chr(i)+ chr(j) + key2, 'ascii')
                #print(key)
                cbc = AES.new(key, AES.MODE_CBC, message[16:])
                C1 = bytes.fromhex(ciphertext1)
                C0prime = cbc.decrypt(C1).hex()
                #print(m1)
                if C0prime[6:8] == 'c2' and C0prime[18:20] == 'f8' and C0prime[28:30] == '2a':
                        print("FOUND KEY")
                        print(C0prime)
                        print(key)
```

```
imacellist@imacellist-VirtualBox: ~/Desktop/Assignment1/aesgoodaesme
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/aesgoodaesme$ python3 aes.py
FOUND KEY
3c8aabc2edfc8afe35f81dacff232a83
b'6F738g9Zzc1S3j4g'
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/aesgoodaesme$
```

Now that we have the key and C0, we can determine flag[1] by decrypting C0 with the key and IV = P0. Since C0 = Ek(P0 XOR flag[1]) => flag[1] = Dk(C0) XOR P0.

```python
keyfound = b'6F738g9Zzc1S3j4g'
i = 99
j = 49

ciphertext0 = '3c8aabc2edfc8afe35f81dacff232a83'


cbc1 = AES.new(keyfound, AES.MODE_CBC, message[:16])
C0 = bytes.fromhex(ciphertext0)
flag1 = cbc1.decrypt(C0)
print(bytes(chr(i)+ chr(j), 'ascii') + flag1)
```

```
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/aesgoodaesme$ python3 aes.py
FOUND KEY
3c8aabc2edfc8afe35f81dacff232a83
b'6F738g9Zzc1S3j4g'
99 49
b'c1pH3r_BLoCk_ch4iN'
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/aesgoodaesme$
```

Appending to flag[0] we found as the 2 unknown bytes of the key, we get the flag CS2107{c1pH3r_BLoCk_ch4iN}.

B.6 Unserialize Hash Length (15 Points)

From the getBadge function, we see that the user has to be at level 2107 to get the flag. In order to add a new user with level 2107, we can use the hash length extension attack to append our new info and modify the cookies.

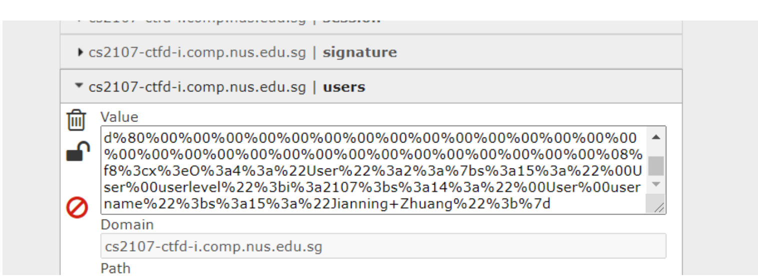
From this article, we see that an application is susceptible to a hash length extension attack if it prepends a secret value to a string (sign function), hashes it with a vulnerable algorithm (SHA-256 is vulnerable), and entrusts the attacker with both the string and the hash (from users and signature in cookies).

Fortunately, there are tools available to exploit the length extension attack. We just need to make sure the information we append follows the same format as for other users.

Format:
%3Cx%3EO%3A4%3A%22User%22%3A2%3A%7Bs%3A15%3A%22%00User%00userlevel%22%3Bi%3A<LEVEL>%3Bs%3A14%3A%22%00User%00username%22%3Bs%3A<NAME_LENGTH>%3A%22<NAME>%22%3B%7D

```
imacellist@imacellist-VirtualBox:~/Desktop/hash_extender$ ./hash_extender -d O%3A4%3A%22User%22%3A2
%3A%7Bs%3A15%3A%22%00User%00userlevel%22%3Bi%3A10%3Bs%3A14%3A%22%00User%00username%22%3Bs%3A8%3A%22
John+Doe%22%3B%7D%3Cx%3EO%3A4%3A%22User%22%3A2%3A%7Bs%3A15%3A%22%00User%00userlevel%22%3Bi%3A33%3Bs
%3A14%3A%22%00User%00username%22%3Bs%3A12%3A%22Peter+Parker%22%3B%7D%3Cx%3EO%3A4%3A%22User%22%3A2%3
A%7Bs%3A15%3A%22%00User%00userlevel%22%3Bi%3A87%3Bs%3A14%3A%22%00User%00username%22%3Bs%3A11%3A%22G
abe+Newell%22%3B%7D --data-format=html -s 5816211e284ab224a1f6988f06f4643006ede4d913a49b352dd0d1dd1
181c207 -a %3Cx%3EO%3A4%3A%22User%22%3A2%3A%7Bs%3A15%3A%22%00User%00userlevel%22%3Bi%3A2107%3Bs%3A1
4%3A%22%00User%00username%22%3Bs%3A15%3A%22Jianning+Zhuang%22%3B%7D --append-format=html -l 32 --ou
t-data-format=html
Type: sha256
Secret length: 32
New signature: 4646a312328e8e1f7fb664a85c8f7e3ee053663c2d14b8a3cd1300b7035625ae
New string: O%3a4%3a%22User%22%3a2%3a%7bs%3a15%3a%22%00User%00userlevel%22%3bi%3a10%3bs%3a14%3a%22%
00User%00username%22%3bs%3a8%3a%22John+Doe%22%3b%7d%3cx%3eO%3a4%3a%22User%22%3a2%3a%7bs%3a15%3a%22%
00User%00userlevel%22%3bi%3a33%3bs%3a14%3a%22%00User%00username%22%3bs%3a12%3a%22Peter+Parker%22%3b
%7d%3cx%3eO%3a4%3a%22User%22%3a2%3a%7bs%3a15%3a%22%00User%00userlevel%22%3bi%3a87%3bs%3a14%3a%22%00
User%00username%22%3bs%3a11%3a%22Gabe+Newell%22%3b%7d%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%08%f8%3cx%3eO%3a4%3a%22User%22%3a2%3a%7bs%3a15%3a%2
2%00User%00userlevel%22%3bi%3a2107%3bs%3a14%3a%22%00User%00username%22%3bs%3a15%3a%22Jianning+Zhuan
g%22%3b%7d
```

With the new signature and users, we can edit the cookie to add ourselves to the Hacker Wall of Fame.

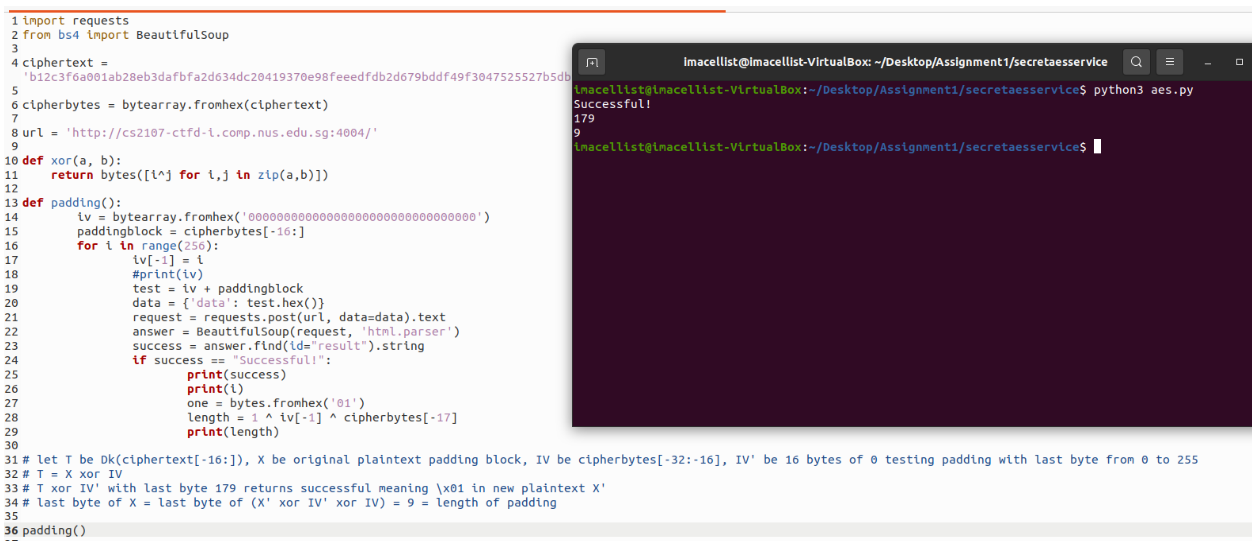


B.7 Secret AES Service (15 Points)

In this Secret AES Service, a valid padding is once which increases from \x01 to \x<length of padding>.

Hence, to determine the length of padding for the last block of plaintext, we can manipulate the IV to the last ciphertext block (C6) so that the last byte of the resulting plaintext = \x01 which is valid.

C6 = Ek(P6 XOR C5) => P6 = Dk(C6) XOR C5
Let IV’ = 000000000000000t where we try all t from 0 to 255
If P6’ = Dk(C6) XOR IV’ returns “Successful!”, it means the padding of P6’ is now \x01
Then last byte of P6 = last byte of (P6’ XOR IV’ XOR C5)



```python
import requests
from bs4 import BeautifulSoup

ciphertext =
'b12c3f6a001ab28eb3dafbfa2d634dc20419370e98feeedfdb2d679bddf49f3047525527b5db

cipherbytes = bytearray.fromhex(ciphertext)

url = 'http://cs2107-ctfd-i.comp.nus.edu.sg:4004/'

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

def padding():
        iv = bytearray.fromhex('00000000000000000000000000000000')
        paddingblock = cipherbytes[-16:]
        for i in range(256):
                iv[-1] = i
                #print(iv)
                test = iv + paddingblock
                data = {'data': test.hex()}
                request = requests.post(url, data=data).text
                answer = BeautifulSoup(request, 'html.parser')
                success = answer.find(id="result").string
                if success == "Successful!":
                        print(success)
                        print(i)
                        one = bytes.fromhex('01')
                        length = 1 ^ iv[-1] ^ cipherbytes[-17]
                        print(length)

# let T be Dk(ciphertext[-16:]), X be original plaintext padding block, IV be cipherbytes[-32:-16], IV' be 16 bytes of 0 testing padding with last byte from 0 to 255
# T = X xor IV
# T xor IV' with last byte 179 returns successful meaning \x01 in new plaintext X'
# last byte of X = last byte of (X' xor IV' xor IV) = 9 = length of padding

padding()
```

```
imacellist@imacellist-VirtualBox: ~/Desktop/Assignment1/secretaesservice
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/secretaesservice$ python3 aes.py
Successful!
179
9
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/secretaesservice$
```

We found that padding is valid when t = 179, hence the last byte of P6 = 1 XOR 179 XOR 187 = 9 = length of padding.

To decrypt each subsequent byte p before the padding, we have to change the padding from \x01 to \x<length of padding> to \x01 to \x<length of padding + 1>.

E.g P = ? ? ? ? ? ? p 01 02 03 04 05 06 07 08 09
P’ = ? ? ? ? ? ? 01 02 03 04 05 06 07 08 09 0a
T = 00 00 00 00 00 t 03 01 07 01 03 01 0f 01 03

Since P’ = Dk(C) XOR IV XOR T = P XOR T where T is what we need to XOR with the current IV to produce the next valid padding, we can try all t from 0 to 255 in the corresponding position to produce \x01. The subsequent p = 1 XOR t.

We can then start decrypting from the last byte for all blocks except the last block which we start from the 7th byte since there are 9 bytes of padding.

```python

def generateNewPadding(length):
        padding = bytearray.fromhex('00000000000000000000000000000000')
        for i in range(17-length, 16):
                curr = (i + length - 16) ^ (i + length - 15)
                padding[i] = curr
        print(padding)
        return padding


# X' = X xor ....t[new padding]

flag = []

def decryptblock(IV, block, start):
        curr = IV
        for i in range(start, 16):
                padding = generateNewPadding(i + 1)
                for j in range(256):
                        padding[15-i] = j
                        iv = xor(curr, padding)
                        test = iv + block
                        data = {'data': test.hex()}
                        request = requests.post(url, data=data).text
                        answer = BeautifulSoup(request, 'html.parser')
                        success = answer.find(id="result").string
                        if success == "Successful!":
                                print(success)
                                print(chr(j^1))
                                flag.insert(0, chr(j^1))
                                curr = iv
                                break

decryptblock(cipherbytes[-32:-16], cipherbytes[-16:], 9)
decryptblock(cipherbytes[-48:-32], cipherbytes[-32:-16], 0)
decryptblock(cipherbytes[-64:-48], cipherbytes[-48:-32], 0)
decryptblock(cipherbytes[-80:-64], cipherbytes[-64:-48], 0)

print(''.join(flag))
```

```
imacellist@imacellist-VirtualBox: ~/Desktop/Assignment1/secretaesservice
imacellist@imacellist-VirtualBox:~/Desktop/Assignment1/secretaesservice$ python3 aes.py
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x03\x01\x07\x01\x03\x01\x0f\x01\x03')
Successful!
}
bytearray(b'\x00\x00\x00\x00\x00\x00\x03\x01\x07\x01\x03\x01\x0f\x01\x03\x01')
Successful!
3
bytearray(b'\x00\x00\x00\x00\x00\x03\x01\x07\x01\x03\x01\x0f\x01\x03\x01\x07')
Successful!
c
bytearray(b'\x00\x00\x00\x00\x03\x01\x07\x01\x03\x01\x0f\x01\x03\x01\x07\x01')
Successful!
1
bytearray(b'\x00\x00\x00\x03\x01\x07\x01\x03\x01\x0f\x01\x03\x01\x07\x01\x03')
Successful!
v
bytearray(b'\x00\x00\x03\x01\x07\x01\x03\x01\x0f\x01\x03\x01\x07\x01\x03\x01')
Successful!
r
bytearray(b'\x00\x03\x01\x07\x01\x03\x01\x0f\x01\x03\x01\x07\x01\x03\x01\x1f')
Successful!
3
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
Successful!
```

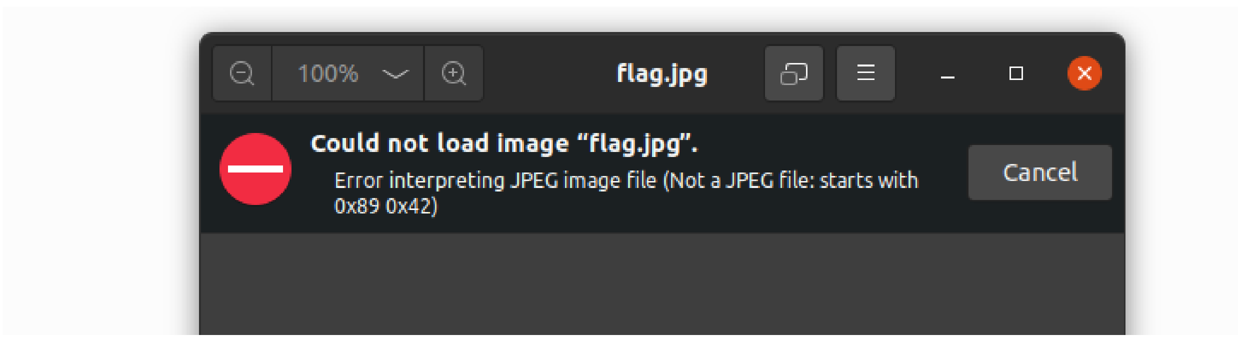
The flag can be found in the last 4 blocks CS2107{1_l1k3_7h15_p4dd1n6_0r4cl3_53rv1c3}.

Section C: Bonus

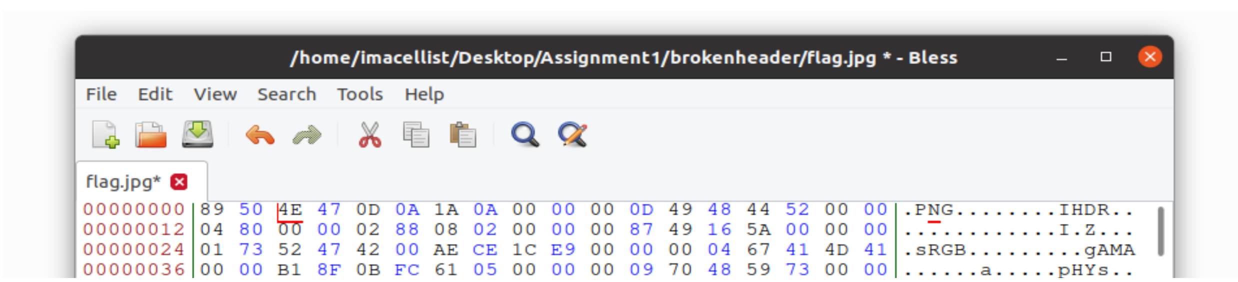
C.1 Broken Headers (5 Points)

Since the prompt asks “What determines an image to be an image?” Maybe we just have to convert it to the appropriate file type.

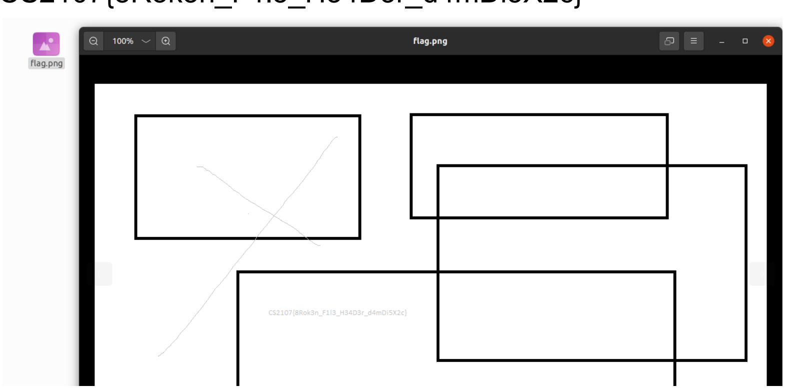
When we try to open it as a .jpg, we get an error message “Error interpreting JPEG image file (Not a JPEG file: starts with 0x89 0x42)”



So the header is really broken. Using a hex editor such as bless, we can change the header from 0x89 0x42 to 0x89 0x50 which renames it to .PNG



Now we can see the flag in the image when opening it as a .png
CS2107{8Rok3n_F1l3_H34D3r_d4mDi5X2c}

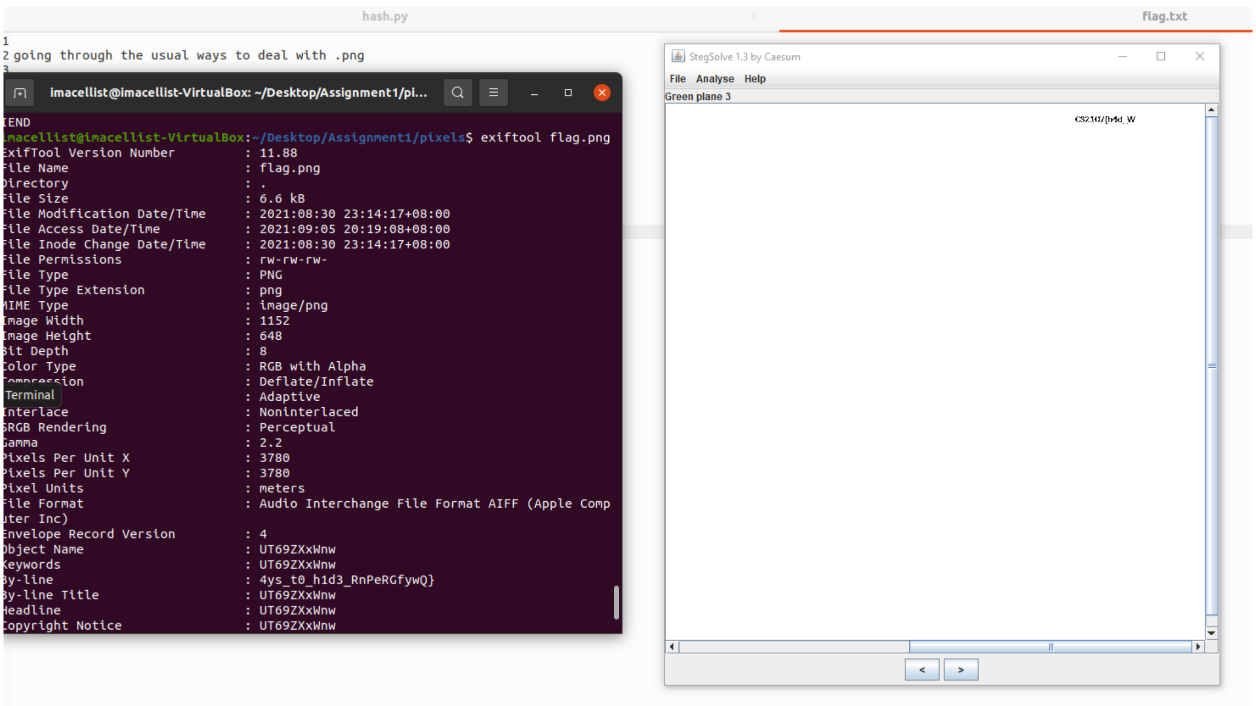


C.2 Pixels (5 Points)

We go through the usual ways to deal with .png files such as file, hexdump, strings etc…

Using exiftools gives us part of what looks like the flag.

The black and white image also indicates it might be a steganography challenge. Using stegsolve and shifting through the different planes, we can make out the start of the flag.



CS2107{b4d_W4ys_t0_h1d3_RnPeRGfywQ}