

# BASE 64

## Description

- The challenge used Base64 encoding to hide the flag, but the base64 characters are converted into emojis. The 64 emojis (key) is provided in the challenge, and this maps to the 64 characters in the Base64 encoding scheme. For instance, 🍌 maps to index 0 which is A, 😊 maps to index 1 which is B, and so on... To get the base64 encoded flag, get the index of each emoji in the flag (😊 is index 16) and then match the index to the Base64 character list (index 16 = Q). Finally, to get the plaintext flag, decode the base64 flag using the b64decode package from Python.

## Insight

- One of the algorithms in the private-key encryption scheme is the *decryption algorithm*. It involves taking a key *k* and a ciphertext *c*, in order to output the plaintext *m* (which in this case is the flag). Since the challenge revealed the 'first 64 emojis' used in encryption, the team realized that its number (64) matches the number of characters in the base64 encoding scheme.

## Solution

- Review the Base64 character chart.

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	ø
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/

2. Using Python, create a list of the given 64 emojis so that each emoji's index is easily retrievable.

```
emojiDB = list("🤔😄😁😂😃😅😆😇😈😉😊😋😌😍😎😏😐😑😒😓😔😕😖😗😘😙😚😛😜😝😞😟😠😡😢😣😤😥😦😧😨😩😪😫😬😭😮😯😰😱😲😳😴😵😶😷😸😹😺😻😼😽😾😿🐉Python  
😡😢😣😤😥😦😧😨😩😪😫😬😭😮😯😰😱😲😳😴😵😶😷😸😹😺😻😼😽😾😿🐉")
```

3. Find the index of each emoji in the encoded flag corresponding to the `emojiDB` list. For instance, the first emoji in the flag 🤨 is index 16 in the `emojiDB` list. Repeat until all emojis have a corresponding index.

[illegible]

4. Use the `b64num` index list in finding the corresponding Base64 character of each decimal number. For example, index 16 (`b64[16]`) is Q. Repeat until all decimal numbers are converted into characters.

```
b64 = list('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=') # from the Python
character chart
b64letters = ''
for num in b64num:
    letters = b64[num]
    b64letters += letters
```

```
#b64letters
'O1NDSV8xOD0uMDNF01RGe2Jhc2U2NCt1bTBqMV9pc19uMHRfZW5jcmlwNzEwbiEhISF9'
```

## Python

5. Decode the string using the Base64 python package.

```
from base64 import b4encode, b4decode
b4decode(b4letters)
```

## Python

```
b'CSFI 184.03 CTF{base64+em0j1 is n0t encryp710n!!!!}'
```

## Python

### Final answer

```
CSCI_184.03_CTF{base64+em0j1_is_n0t_encryp710n!!!!}
```

# Shattered

## Description

"This industry cryptographic hash function standard is used for digital signatures and file integrity verification ([shattered.io](https://shattered.io)).\" In 2017, cryptographic researchers announced the first real-world collision attack that allows the creation of two documents with different contents, but have the same SHA-1 digital signature.

In this challenge, the flag is revealed when SHA-1 digests of the username and password are the same, but the caveat is that its input values must be different. To solve the challenge, the user must find two unique input values that have the same SHA-1 digest, then send it to the login server through the POST method.

## Insight

The webpage shows a simple sign-in form. Upon trying out different login combinations, `admin:admin`, `root:admin`, `admin:password`. The group identified two main responses:

- **Error:** Your password cannot be the same as your username.'
- **Error:** Incorrect password.'

This is also inline with the responses from `admin_server.py` code.

Upon further inspection of the Network tab in the browser's developer tools, the payload shows that the login page encodes the username and password using Base64. Afterwards, the group was able to identify the variables used in the form data (`user` & `pass`) that is submitted to the backend. For example for `admin:admin`, the parsed form data is `user=YWRtaW4%3D&pass=YWRtaW4%3D`.

After inspecting the webpage, the group viewed the `admin_server.py` back-end code. From here, we saw that:

1. The server uses the POST method to pass the data.
2. The server decodes the data using a Base64 decoder.
3. **The flag is revealed if the username and password are different, but their SHA-1 digest are the same.**

Going back to the form responses, the **username and password cannot be the same**; therefore, in order to get the flag, the user must input a username and password combination that have *different* values but have the *same* SHA-1 digest.

The website, <https://shattered.io/>, discusses that it is now possible for attackers to produce two different documents with same SHA-1 digital signature (collision attack). Therefore, we use the same SHA-1

## Solution

1. Find two inputs that have the same exact values but with different SHA-1 digest.
  - The website, <https://shattered.io/>, provides two PDF files that have different content but same SHA-1 digest.
  - PDF links: <https://shattered.io/static/shattered-1.pdf> & <https://shattered.io/static/shattered-2.pdf>

- Using Python, open PDF links and read its content. Store the content of the 1st PDF as the username `admin`, then store the content of the 2nd PDF as the password `adminpw`. In order to not exceed the max length data that the POST method can handle, we'll only use the first 1000 lines.

```
from urllib.request import urlopen
```

Python

```
admin = urlopen("http://shattered.io/static/shattered-1.pdf").read()[0:1000]
adminpw = urlopen("http://shattered.io/static/shattered-2.pdf").read()[0:1000]
```

- From what we saw on the payload from the Networks tab, the log-in form encodes the username and password using Base64 before it passes the data to the back-end server.

```
from base64 import b64encode
```

Python

```
bfwuser = b64encode(admin)
bfwpass = b64encode(adminpw)
```

- Send a POST request to the login page using the variables from the payload and the Base64 encoded credentials from the previous step.

```
import requests
```

Python

```
url = 'http://bfwadmin.lunchtimeattack.wtf/login'
myjson = {'user': bfwuser, 'pass': bfwpass}

req = requests.post(url, data = myjson)
req.text
```

```
'CSCI_184.03_CTF{7h3_SHA_1n_sha-1_57and5_4_SHAtt3r3d!!!}'
```

Python

## Final answer

```
CSCI_184.03_CTF{7h3_SHA_1n_sha-1_57and5_4_SHAtt3r3d!!!}
```

# I Wanna Cry

## Description

The challenge provides a ZIP file containing encrypted files with the extension `.enc`. To find the flag, the user must decrypt the file `flag.txt.enc`. From the provided encryption code, the team identified that the encryption uses AES in CTR mode. Additionally, since we have a plaintext-ciphertext pair from `aeslocker.py` and `aeslocker.py.enc`, then it's possible to XOR the given pair with the ciphertext flag to decrypt it and get the plaintext.

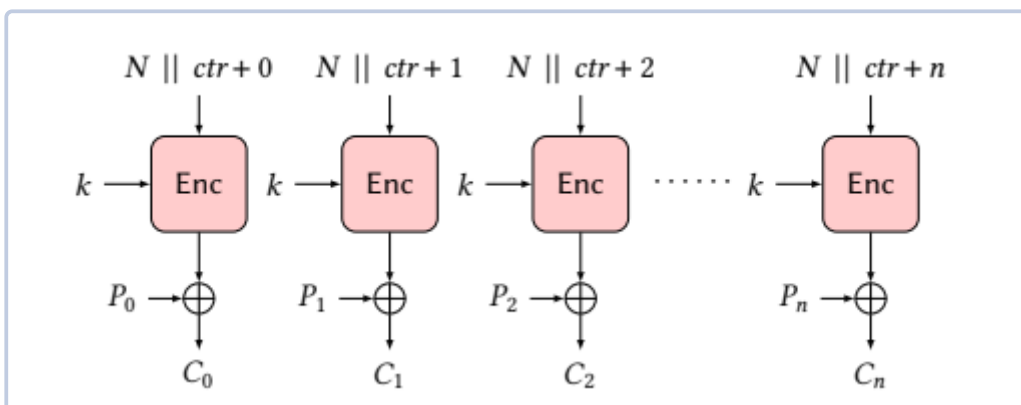
## Insight

The AESLocker ransomware is a Python script that encrypts all files in a folder using the AES-CTR mode `Cipher(algorithms.AES(key), modes.CTR(nonce))`. Once the ransomware is executed, it replaces the data inside each file with the ciphertext. After replacing the data in the first file, the counter increases by 1; then the script repeats the encryption until all files in a folder are encrypted.

Advanced Encryption Standard-Counter Mode (AES-CTR) "essentially turns a block cipher into a stream cipher." [CSCI 184.03 Lecture 9: Block Ciphers](#). Encryption using the CTR mode uses a keystream created from two values (key and nonce) then XOR'ed with the plaintext.

"The keystream block is generated by  $K_i := \text{Enc}_k(N || \text{ctr} + i)$  and encryption is done by ciphertext  $C_i := P_i \oplus K_i$ ."

The figure ([CSCI 184.03 Lecture 9: Block Ciphers](#)) below illustrates how encryption works in AES-CTR:



The challenge provides a ZIP file containing the encrypted files. One of the encrypted files is the actual ransomware script `aeslocker.py.enc`, and since we have the contents of the script `aeslocker.py`, we can use these two as the ciphertext-plaintext pair to get the keystream block.

We can exploit that the script contains an **IV and the key that are generated only once in encrypting each file**. Therefore, it's possible to XOR two ciphertext to get the plaintexts.

$$C_1 \oplus C_2 = P_1 \oplus K \oplus P_2 \oplus K$$

Then, since we have a ciphertext-plaintext pair, we can XOR these with the ciphertext flag to get the plaintext flag.

$$P_2 = C_1 \oplus C_2 \oplus P_1$$

## Solution

As explained in the Insight section, we can XOR  $C_1$ ,  $C_2$ ,  $P_1$  to get  $P_2$ . In the case of the challenge:

- $C_1 := \text{aeslocker.py.enc} := \text{aes\_c}$
- $P_1 := \text{aeslocker.py} := \text{aes\_p}$
- $C_2 := \text{flag.txt.enc} := \text{flag\_c}$
- $P_2 := \text{flag.txt} := \text{flag\_p}$

Since the script uses AES-CTR mode with the key length and nonce length of 16 bytes, the encryption is also done by 16-byte blocks. After the encryption of each file, the value of IV is incremented by 1.

To create the blocks, we use the code below provided from one of the homeworks:

```
def create_blocks(enc, n):  
    blocks = []  
    for i in range(0, len(enc), n):  
        blocks.append(enc[i:i+n])  
    return blocks
```

Python

Then we split the ciphertexts and plaintexts using the `create_blocks` function:

```
aes_p = create_blocks(open('aeslocker.py', 'rb').read(), 16)  
aes_c = create_blocks(open('aeslocker.py.enc', 'rb').read(), 16)  
flag_c = create_blocks(open('flag.txt.enc', 'rb').read(), 16)
```

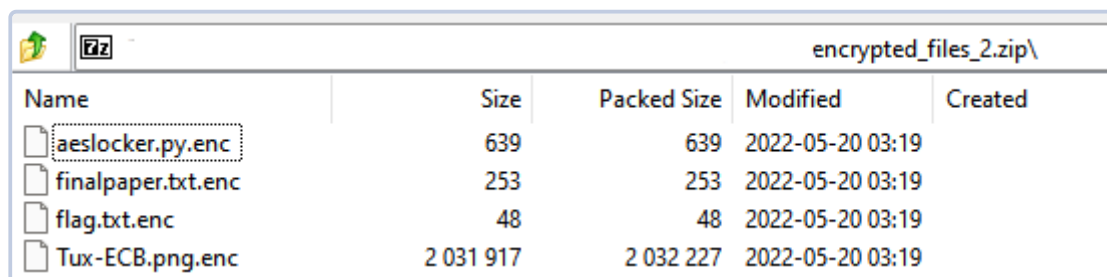
Python

As mentioned in the Insight part, the plaintext flag can be generated using the XOR of the `aeslocker.py` plaintext and ciphertext pair, and the `flag.txt.enc` ciphertext.

```
# since  
flag_p[0] ^ flag_c[0] = aes_p[count] ^ aes_c[count]  
# therefore:  
flag_p[0] = aes_p[count] ^ aes_c[count] ^ flag_c[0]
```

Python

And since there are four files, the only possible increment (`count`) is upto 3.



Name	Size	Packed Size	Modified	Created
aeslocker.py.enc	639	639	2022-05-20 03:19	
finalpaper.txt.enc	253	253	2022-05-20 03:19	
flag.txt.enc	48	48	2022-05-20 03:19	
Tux-ECB.png.enc	2 031 917	2 032 227	2022-05-20 03:19	

```
for count in [1, 2, 3]:  
    result = ''  
    for i in range(len(flag_c)):
```

Python

```

xor_aes = xor_bytes(aes_p[i + count], aes_c[i + count]) # keystream
xor_flag = xor_bytes(flag_c[i], xor_aes) # keystream xor flag_c
result += ''.join(map(chr, xor_flag))
print(count, ' - ', result)

```

Once we execute the code, it will return 3 lines, and one of those lines return the flag.

```

1 - (ÿàª.»   Ûû}  Ě4rÔÔð <Ü-®)âù$H  ÝÀ kG p SãØ lÂ
2 - ÿ Íò$    gÕóCg ¶Î ZK| h  ~ U${KD}õ?0j49îè Ê~¥
3 - CSCI_184.03_CTF{n0nc3_r3us3__str1ke5_y3t_4g41n!}

```

Python

## Final answer

```
CSCI_184.03_CTF{n0nc3_r3us3__str1ke5_y3t_4g41n!}
```

# RSA Encryption

## Description

The RSA encryption scheme is insecure, but a huge stepping stone for more secure schemes that are used in practice (Guadalupe, 2022). It makes use of public and private keys, wherein the concept is somewhat similar to giving someone you trust the key to your house to give them access to your place.

The challenge used 256-bit RSA encryption to hide the flag. The details given are the modulus  $N$ , public exponent  $e$  and ciphertext  $c$ .

## Insight

In this situation, there was no vulnerability in the situation, the situation was more of incorrectly generated RSA keys, wherein the problem was that it is impossible to decrypt due to the incorrectly generated keys. When generating the public exponent,  $e$ , this must be relatively prime to Euler's totient function. The problem is that this step was not assured in this situation. Lucky for us, there was an algorithm created to decrypt incorrectly generated  $p$  and  $q$  primes in an RSA scheme. The algorithm recovers potential plaintexts in seconds for 4096-bit keys.

## Solution

### 1. Understanding the problem

Initially, the problems that the group was looking for were weakly generated RSA keys, or small generated RSA keys. This was done by trying to factor out modulus  $N$ , and double checking the size of modulus  $N$ . In this case, none of the aforementioned were the problems, instead, the problem was trying to recover the plaintext because of the incorrectly generated keys. In the case of incorrectly generated keys, it is impossible to decrypt the message.

The keys were incorrectly generated because they oversaw the step that the public exponent  $e$  must be relatively prime to totient's function. In this case, totient's function is divisible by  $e$ .

```
# Double checking e

totient = (p - 1) * (q - 1)
print(totient % e)

✓ 0.5s

0
```

### 2. Creating Algorithm 1

After understanding the problem, we identified that the decryption algorithm was taken from [Daniel Shumow's paper on "Incorrectly Generated RSA Keys"](#). Algorithm 1 makes use of a subgroup  $E$ , which is a set of  $e$ -order elements in  $(\mathbb{Z}/N\mathbb{Z})^X$  that will give all



the potentially lost factors of the plaintext.

**Algorithm 1** Find multiplicative generator  $g_E$  of the subgroup of  $e$  order elements  $E < (\mathbb{Z}/N\mathbb{Z})^\times$ .

```
 $\tilde{\varphi} \leftarrow (p-1)(q-1)/e$   
 $g \leftarrow 1$   
repeat  
   $g \leftarrow g + 1$   
   $g_E \leftarrow g^{\tilde{\varphi}} \bmod N$   
until  $g_E \neq 1$   
return  $g_E$ 
```

The end result of the algorithm is returning a generator  $g_E$  of  $E$ . This generator is responsible for generating all elements of the subgroup  $E$ , which is important to use in Algorithm 2.

```
# Algorithm 1  
def rsa_decrypt(c, N, e, p, q):  
    totient = (p-1) * (q-1)  
    totient_over_e = totient / e  
    g = 1  
    ge = 1  
  
    while ge == 1:  
        g += 1  
        ge = pow(g, totient_over_e, N)  
  
    print(f'GE: {ge}')
```

Python

### 3. Creating Algorithm 2

Algorithm 2 makes use of this generator to create all elements from subgroup  $E$ , and using each element, creates a plaintext with correct padding. The algorithm then returns all potential plaintexts from the code.

**Algorithm 2** Find set of potential plaintexts that encrypt to ciphertext  $c$  with incorrectly generated RSA key  $N = p \cdot q$  and public exponent  $e$ , given prime factor  $p$  and  $q$ .

$$\tilde{\varphi} \leftarrow (p-1)(q-1)/e$$
$$d \leftarrow e^{-1} \bmod \tilde{\varphi}$$
$$a \leftarrow c^d \bmod N$$

Use Algorithm 1 to find multiplicative generator  $g_E$  of  $e$ -order elements of  $(\mathbb{Z}/N\mathbb{Z})^\times$ .

$$P \leftarrow \{\}$$
$$\ell \leftarrow 1 \bmod N$$

**for all**  $i = 0 \cdots e - 1$  **do**

$$x \leftarrow a \cdot \ell \bmod N$$

**if**  $x$  is a correctly padded plaintext. **then**

$$P \leftarrow P \cup \{x\}$$

**end if**

$$\ell \leftarrow \ell \cdot g_E \bmod N$$

**end for**

**return** Set  $P$  of potential plaintexts.

Due to time constraints, we were unable to implement Algorithm 2 fully. Instead of looking for the correct padding, we decided to go through all the plain texts and figure out which of the plain texts looks most likely to be the flag. We faced a problem here, however, because some of the plain texts returned a non-hexadecimal number, giving an error. In order to bypass this, the group used a try and except clause so that the algorithm would continue running despite running through an error. From there, we got the flag: `CSCI_184.03_CTF{n0_m3s5age_15_und3cr4p7ab13}`

```
# Algorithm 2
p = []
plain_texts = []
d = mod(1/e, totient_over_e)
a = pow(Integer(c), d, N)
ell = mod(1, N)

print(f'd: {d}')
print(f'a: {a}')

for i in range(0, e):
    x = mod(a * ell, N)
    p.append(x)
    ell = mod(ge * ell, N)

for j in p:
    try:
        plain_texts.append(bytes.fromhex(hex(j).replace('0x', '')).decode('unicode-escape'))
    except Exception as e:
        print(e)

return plain_texts
```

Python

## Outputs:

```
GE: 253901037683708185681386310023172598050636236040256731583903659
d: 3820632164638236978737311148464568546239240969048632384943208034
a: 2342578193413403400256394147816138761795865313784080596101003316
non-hexadecimal number found in fromhex() arg at position 127
non-hexadecimal number found in fromhex() arg at position 127
non-hexadecimal number found in fromhex() arg at position 127
non-hexadecimal number found in fromhex() arg at position 127
non-hexadecimal number found in fromhex() arg at position 127
non-hexadecimal number found in fromhex() arg at position 127
```

```
[',°H\xa0\x07E\xad\x9e\x00\x97Î\x14³êÁ\x97\x06C`ýfkaP`?ã°\x0eG\x10!\x1fÂKÔMPJTæ\x99;%øqÁo\x1dñOñ\
'J|+óCE\x1a~lè;ü"Ùæe\'. _xü:\x84NÃü«²ã\x80%áBT\r+4[÷*Ü\x1e\n\x1b:R\x15\x9e\x1açŸW\x90ç\x02,4\x01%â
'*b1||g\x91-\x9dÛRÊh.{\x8eç\x84f#eÖç\x96Ai%f§n\x1ceùEt²é\x9d\x1aÌ\x95çË/Ñô{ú«` \x82ÊÖãÖ\x1fÚP e{k\
'.zHëI\r@\x89c\x1dö)uç\x91ã9ß\x19`[\x8f_ø;¹p\x9d\x88CÛ}\x1bÃ2\x7fâZß\x7f\x98ÚÁ³\x9c»ç-Ú\x9d\x07»
'A0\x08\x9d\x15ô\x96m\x81tgRØ\x9d\x14qk\x07B\x04\x14$%j\x0cLôQ\x9c%\x89\x1f0\r°\x0cytvF\x13)ffH^w
",I7û'\x9c\x89n\x82ē./Îøúó_\x18aC\x96öI^äsò\x05ûē<³é\r%Î¥æüPÎÖ$^\x14R\x192î¹Ùêi{[\x8bõ]fµMå\x1e\x
'\x15\x00\x88\x84:\x12\x92B«¢Bc"eá%\t\x19ji)\x82²%Õn:\x10\x9d\x97\x04<ibó)þUþðíYç ÎÃİp\x84ãC\x8e\x
'(s¢Á\x9bê\x0bá;ÎK9µs¹\x86ÎºT\x8cçã\x8a\x05\x1ax% \x9dÕn\x82àòN;í;`ò\x81c` .â}A0Ã`È\x97ü\x8b|<â\x8b
'CSCI_184.03_CTF{n0_m3s5age_15_und3cr4p7ab13}',
'3ÎG\x17\x94\x0e\x16?\x91A\x03Jlû%ágN\x12Âô:Ñ\x81Ê\x93B9gâkhêkπ*?xû\x14!+\x12g-ø}*ð\x9d\x1fföY\r
'5²»²\x7f¹*fèD\x1d\rî\x11÷5jÝ÷%l\x95+.b`Í5ÛB3³\x93iñèl_\x97eùú\x8dQT6X\nÎÛyîÖ\x812~\x98%«xµ]xò' ]
```

## Final answer

CSCI\_184.03\_CTF{n0\_m3s5age\_15\_und3cr4p7ab13}

# Two-Time Pad

## Description

The Two-Time Pad challenge makes use of the one time pad encryption technique to produce multiple cipher texts using the same key. The one time pad is an uncrackable encryption technique, if multiple conditions are met. In order to encrypt a message, a random secret key is used, and each bit or character of the plain text is encrypted by combining it with the corresponding bit or character of the secret key.

## Insight

The main vulnerability that the group was able to exploit was the use of the same key to encrypt multiple cipher texts. Hence, it's possible to XOR two ciphertext to get the plaintexts as the key is cancelled out in the equation.

$$C_1 \oplus C_2 = P_1 \oplus K \oplus P_2 \oplus K$$

In this situation, it is possible to get the message through the crib drag method. By applying XOR to multiple cipher texts, we were able to uncover small clues of the message. We can get the message by using common words and intelligently guessing the rest of the message based on what is given. This was implemented more easily through the use of a [crib drag calculator](#). (This is explained more in the solutions part.)

Then, after getting the ciphertext-plaintext pair, we can XOR these with the ciphertext flag to get the plaintext flag.

$$P_2 = C_1 \oplus C_2 \oplus P_1$$

## Solution

Our first impression when facing this problem was to try and find the key through the use of repeating key XOR. Due to time constraints, however, we were unable to implement this algorithm, and decided to use the crib drag method since we knew a decent amount of characters already, `CSCI_184.03_CTF{`.

The **strategy** was to look for the remaining characters of "CSCI\_184.03\_CTF{". In order to do this, we focused on using 3 ciphertexts: C1, C2, and C5 (flag).

We knew that the characters were in C5, so we applied XOR to C2 and C5 to retrieve the first 16 characters of M2. Image below is from the crib drag website (using C2 as ciphertext 1 input, and flag as ciphertext 2 input). The result from the crib drag is the message from C2 which is `here is another`.

Ciphertext1 (Hex):

781c0443cc2852fb7b865792674824880  
952db6a676c6752514f37d1bb3e20713f  
0ee00294b4fab12fcd88b83f104791559  
bc3fda2081cd2f7e3a4

Ciphertext2 (Hex):

532a356fb37019ef34d80bb94c7910d32  
46cda606d5e335659591bddb93a6e670  
e19fa1ebbabfaac50d392932243569d6fd  
4def0b63253d7f1fff7

Character Set: a-zA-Z0-9,?! ;:'

Click me to show the details

Output 1:

Output 2:

Crib words:

CSCI\_184.03\_CTF

Result:

- here is another

output1

output2

From there, we were able to find the rest of the characters of M2 by using the crib text that we had and the XOR of C1 and C2. By using the previous message, we were able to get some of the plaintext message from C1 which is `this is a random`.

Ciphertext1 (Hex):

64111f55cc2852fb7bc84a87614939c55b  
5ed07d7b60205a14592ad3ae22706034  
09b20c8db4f7f57bd284ec385e56d444d  
2ddf9ef1d5dddbcb4a4

Ciphertext2 (Hex):

781c0443cc2852fb7b865792674824880  
952db6a676c6752514f37d1bb3e20713f  
0ee00294b4fab12fcd88b83f104791559  
bc3fda2081cd2f7e3a4

Character Set: a-zA-Z0-9,?! ;:'

Click me to show the details

Output 1:

Output 2:

Crib words:

here is another

Result:

- this is a random

output1

output2

After that, we tried guessing the plaintext message of C1 and C2 alternately. For instance, we guessed that the next word after `this is random` is `message`. Then by using

the plaintext of C1 as `this is a random message`, we were able to get a longer plaintext of C2 which is `here is another random m`.

Repeatedly, we used `here is a another random m` as the crib words, then we get `this is a random message encry`. We did the alternating guess strategy repeatedly until we got plaintext M2 from C2: `here is another random message encrypted with the same key`.

After retrieving M2, we were able to retrieve M5 by simply applying XOR:  $M2 \oplus C2 \oplus C5$ .

```
from binascii import hexlify, unhexlify Python

# from homework 1
def xor_bytes(a, b):
    xorlist = []
    for (a_ch, b_ch) in zip(a, b):
        xor = a_ch ^ b_ch
        xorb = bytes([xor])
        xorlist.append(xorb)
    xorbytes = b''.join(xorlist)
    return xorbytes

C1 =
unhexlify(b'64111f55cc2852fb7bc84a87614939c55b5ed07d7b60205a14592ad3ae2270603409b20c8db4f7f57bd28
4ec385e56d444d2ddf9ef1d5dddbcb4a4')

C2 =
unhexlify(b'781c0443cc2852fb7b865792674824880952db6a676c6752514f37d1bb3e20713f0ee00294b4fab12fcd8
8b83f104791559bc3fda2081cd2f7e3a4')

C5flag =
unhexlify(b'532a356fb37019ef34d80bb94c7910d3246cda606d5e335659591bddb93a6e670e19fa1ebbabfaac50d39
2932243569d6fd4def0b63253d7f1fff7')

xor_c = xor_bytes(C2, C5flag)

# we already know the first few characters of the flag is CSCI_184.03_CTF{ (len=16)
# use the crib drag calculator online to try and guess the message
https://toolbox.lotusfa.com/crib_drag/

M2 = b'here is another random message encrypted with the same key.'

flag = xor_bytes(M2, xor_c)
flag
```

This gave us a flag of `CSCI_184.03_CTF{__one_time_means_the_key_is_used_only_once}`.

## Final answer

```
CSCI_184.03_CTF{__one_time_means_the_key_is_used_only_once}
```

# Sussy Curve

## Description

**Elliptic Curve Cryptography** is an encryption algorithm that paved the way for stronger security with smaller key sizes. Compared to algorithms like RSA or Diffie Hellman, it is more efficient because of its ability to produce smaller key sizes without sacrificing security.

The Sussy Curve challenge makes use of *elliptic curve construction*, which is the set of solutions to  $y^2 = x^3 + Ax + B$ , where the constants A and B must satisfy  $4A^3 + 27B^2 \neq 0$  to ensure  $x^3 + Ax + B = 0$  has no repeated roots. The given elements in this challenge are prime,  $p$ , elliptical curve,  $E$ , and points P and Q.

## Insight

Due to time constraints, it was not possible for the team to capture the flag for this challenge. However, the attack we were planning to look into using was the invalid-curve attack. Assuming that the input points were not validated when applying the Elliptic curve Diffie-Hellman attack, it is possible to retrieve the shared secret,  $bh_A$ .

## Solution

N/A

## Final answer

N/A