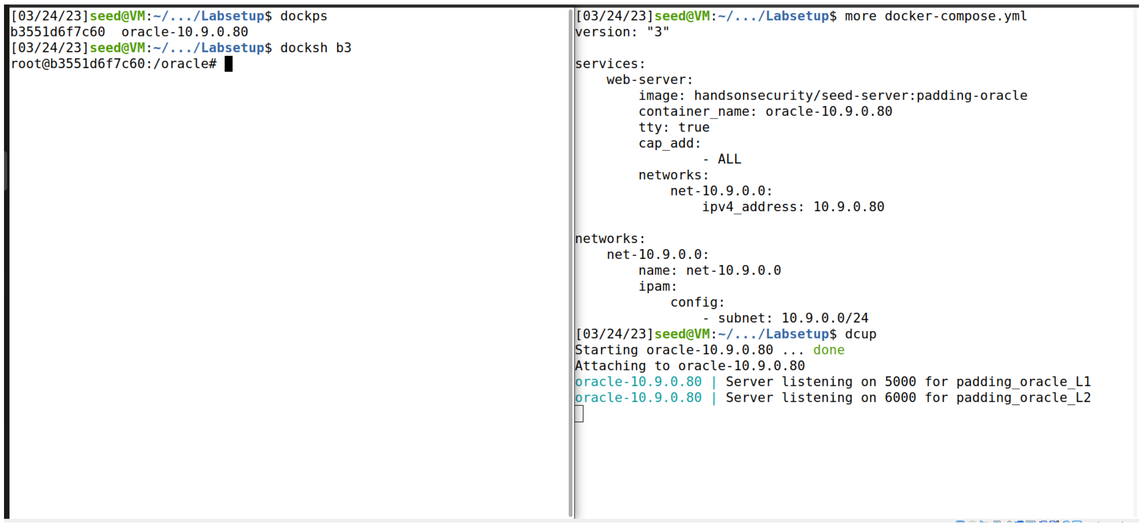
Computer Security **Name:** Ameer Hamza
Assignment#4: Padding Oracle Attack Lab

1. **Initial setup:** Here is the initial setup for the lab.

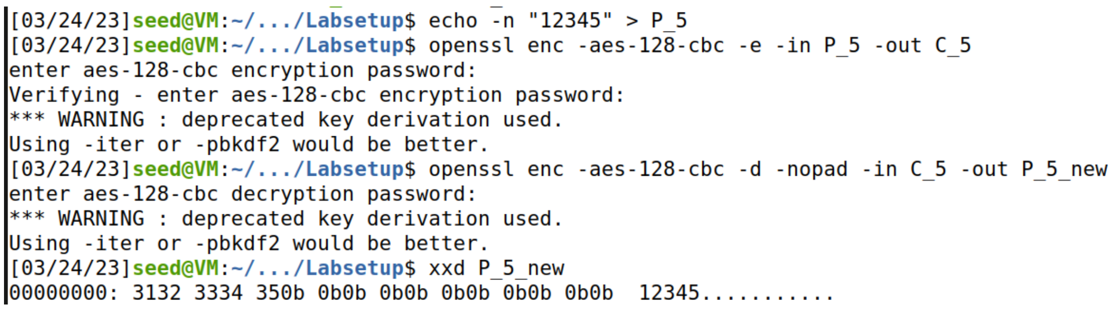


```
[03/24/23]seed@VM:~/.../Labsetup$ dockps
b3551d6f7c60  oracle-10.9.0.80
[03/24/23]seed@VM:~/.../Labsetup$ docksh b3
root@b3551d6f7c60:/oracle# 
```

```
[03/24/23]seed@VM:~/.../Labsetup$ more docker-compose.yml
version: "3"

services:
    web-server:
        image: handsonsecurity/seed-server:padding-oracle
        container_name: oracle-10.9.0.80
        tty: true
        cap_add:
                - ALL
        networks:
            net-10.9.0.0:
                ipv4_address: 10.9.0.80

networks:
    net-10.9.0.0:
        name: net-10.9.0.0
        ipam:
            config:
                - subnet: 10.9.0.0/24
[03/24/23]seed@VM:~/.../Labsetup$ dcup
Starting oracle-10.9.0.80 ... done
Attaching to oracle-10.9.0.80
oracle-10.9.0.80 | Server listening on 5000 for padding_oracle_L1
oracle-10.9.0.80 | Server listening on 6000 for padding_oracle_L2
```

2. **Task 1: Getting Familiar with Padding:**

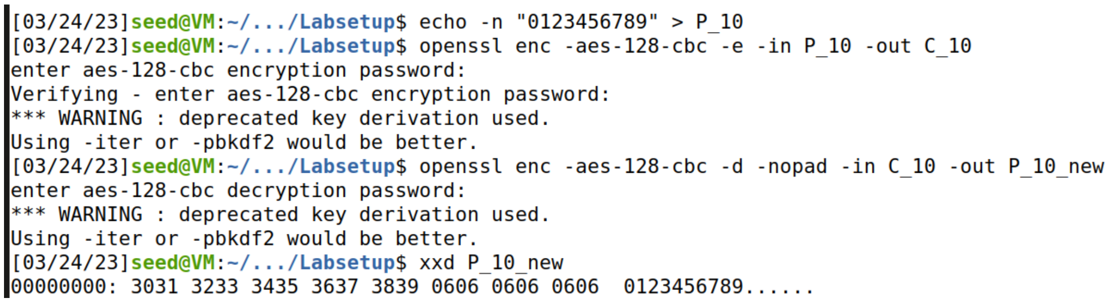
5 bytes file:



```
[03/24/23]seed@VM:~/.../Labsetup$ echo -n "12345" > P_5
[03/24/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -e -in P_5 -out C_5
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[03/24/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -d -nopad -in C_5 -out P_5_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[03/24/23]seed@VM:~/.../Labsetup$ xxd P_5_new
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
```

Two make the length of the content 16, the remaining 11 bytes are padded with 0x0b (11 in decimal).

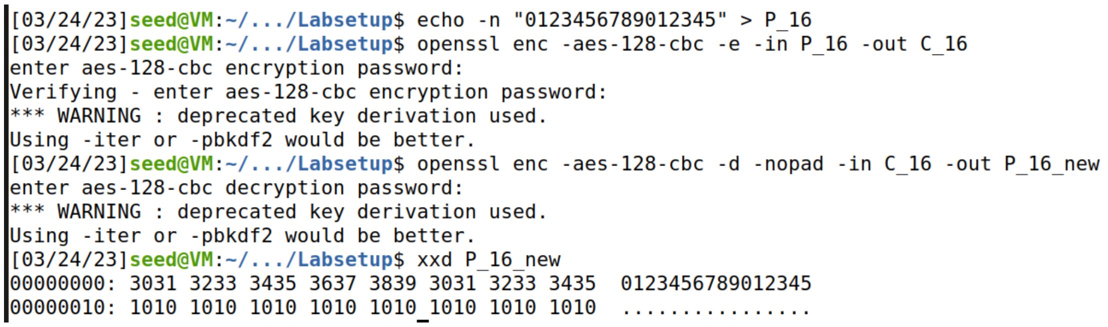
10 bytes file:



```
[03/24/23]seed@VM:~/.../Labsetup$ echo -n "0123456789" > P_10
[03/24/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -e -in P_10 -out C_10
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[03/24/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -d -nopad -in C_10 -out P_10_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[03/24/23]seed@VM:~/.../Labsetup$ xxd P_10_new
00000000: 3031 3233 3435 3637 3839 0606 0606 0606  0123456789......
```

The remaining 6 bytes are padded with 0x06 (6 in decimal).

16 bytes file:

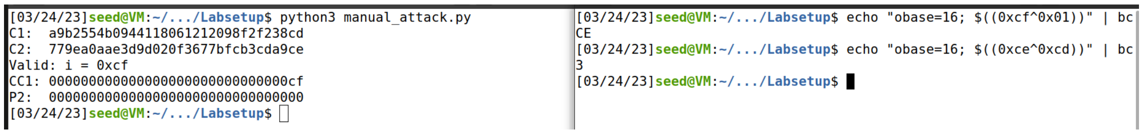


```
[03/24/23]seed@VM:~/.../Labsetup$ echo -n "0123456789012345" > P_16
[03/24/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -e -in P_16 -out C_16
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[03/24/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -d -nopad -in C_16 -out P_16_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[03/24/23]seed@VM:~/.../Labsetup$ xxd P_16_new
00000000: 3031 3233 3435 3637 3839 3031 3233 3435  0123456789012345
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
```

Since 16 is a multiple of 16, no remaining bytes are needed to complete 16 bytes. However, in this case, another 16 bytes are padded with 0x10 (16 in decimal) to avoid some rare cases (last one byte is 0x01, last two bytes are both 0x02, last three bytes are all 0x03, and so on, in the original content).

3. **Task 2: Padding Oracle Attack (Level 1):**

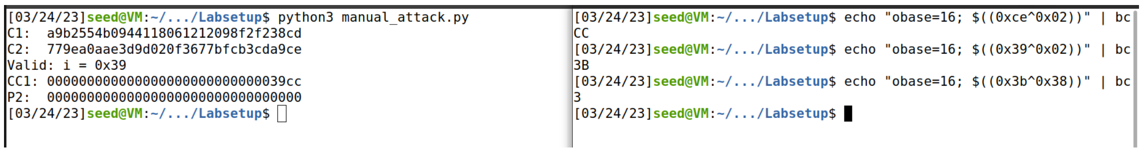
   I learned about the padding oracle from [1]. We start with initial values of `CC1` as all zeros. We first consider all 256 possible values for the last byte `CC1[15]`, and for each, construct `IV+CC1+C2` and send to the oracle. Let's say the second block of corresponding plaintext generated with ciphertext `C2`, decrypted text `D2`, and XORed with `CC1` is `PP2`. The value of `CC1[15]` that gives a valid padding is `0xcf`. Since it is a valid padding, the value for `PP2[15]` must be `0x01`. We find `D2[15] = CC1[15]` $\oplus$ `PP2[15] = 0xcf` $\oplus$ `0x01 = 0xce`. We can compute the last byte of actual plaintext by `P2[15] = D2[15]` $\oplus$ `C1[15] = 0xce` $\oplus$ `0xcd = 0x03`.



```
[03/24/23]seed@VM:~/.../Labsetup$ python3 manual_attack.py
C1:  a9b2554b0944118061212098f2f238cd
C2:  779ea0aae3d9d020f3677bfcb3cda9ce
Valid: i = 0xcf
CC1: 000000000000000000000000000000cf
P2:  00000000000000000000000000000000
[03/24/23]seed@VM:~/.../Labsetup$
```

```
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xcf^0x01))" | bc
CE
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xce^0xcd))" | bc
3
[03/24/23]seed@VM:~/.../Labsetup$
```

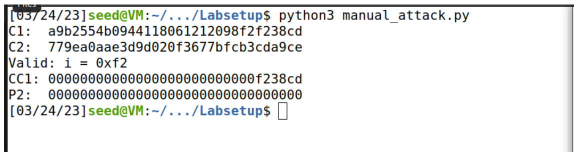
   We need to find `D2[14]`. Hence, we now need to construct another `CC1` that will give `D2[14]`. We choose `CC1[15]` to be a byte such that we get `PP2[15]=0x02` (because now we want to find second last byte and both bytes in plaintext should be padded with `0x02`). We find `CC1[15] = D2[15]` $\oplus$ `0x02 = 0xce` $\oplus$ `0x02 = 0xcc`. We set `CC1[15] = 0xcc` in the attack program, `K=2` and run again. We find `CC1[14] = 0x39`. We find `D2[14] = CC1[14]` $\oplus$ `PP2[14] = 0x39` $\oplus$ `0x02 = 0x3b` (because `PP2[14] = 0x02`). We can compute the byte at 14 index of actual plaintext by `P2[14] = D2[14]` $\oplus$ `C1[14] = 0x3b` $\oplus$ `0x38 = 0x03`.



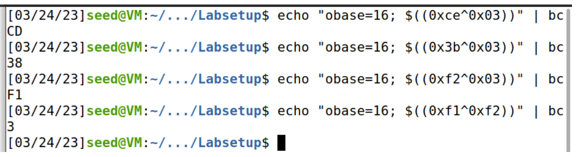
```
[03/24/23]seed@VM:~/.../Labsetup$ python3 manual_attack.py
C1:  a9b2554b0944118061212098f2f238cd
C2:  779ea0aae3d9d020f3677bfcb3cda9ce
Valid: i = 0x39
CC1: 000000000000000000000000000039cc
P2:  00000000000000000000000000000000
[03/24/23]seed@VM:~/.../Labsetup$
```

```
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xce^0x02))" | bc
CC
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x39^0x02))" | bc
3B
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x3b^0x38))" | bc
3
[03/24/23]seed@VM:~/.../Labsetup$
```

   We need to find `D2[13]`. Hence, we now need to construct another `CC1` that will give `D2[13]`. We choose `CC1[15]` and `CC1[14]` to be bytes such that we get `PP2[15]=0x03` and `PP2[14]=0x03` (because now we want to find third last byte and last 3 bytes in plaintext should be padded with `0x03`). We find `CC1[15] = D2[15]` $\oplus$ `0x03 = 0xce` $\oplus$ `0x03 = 0xcd`. We find `CC1[14] = D2[14]` $\oplus$ `0x03 = 0x3b` $\oplus$ `0x03 = 0x38`. We set `CC1[15] = 0xcd` and `CC1[14] = 0x38` in the attack program, `K=3` and run again. We find `CC1[13] = 0xf2`. We find `D2[13] = CC1[13]` $\oplus$ `PP2[13] = 0xf2` $\oplus$ `0x03 = 0xf1` (because `PP2[13] = 0x03`). We can compute the byte at 13 index of actual plaintext by `P2[13] = D2[13]` $\oplus$ `C1[13] = 0xf1` $\oplus$ `0xf2 = 0x03`.

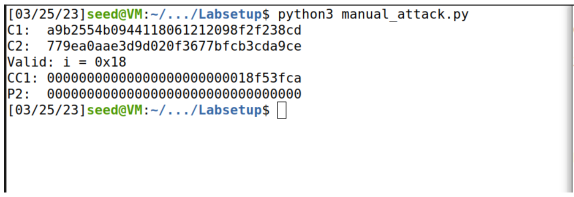


```
[03/24/23]seed@VM:~/.../Labsetup$ python3 manual_attack.py
C1:  a9b2554b0944118061212098f2f238cd
C2:  779ea0aae3d9d020f3677bfcb3cda9ce
Valid: i = 0xf2
CC1: 000000000000000000000000f238cd
P2:  00000000000000000000000000000000
[03/24/23]seed@VM:~/.../Labsetup$ 
```

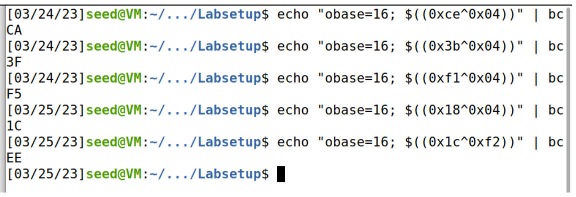


```
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xce^0x03))" | bc
CD
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x3b^0x03))" | bc
38
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xf2^0x03))" | bc
F1
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xf1^0xf2))" | bc
3
[03/24/23]seed@VM:~/.../Labsetup$ 
```

We need to find `D2[12]`. Hence, we now need to construct another `CC1` that will give `D2[12]`. We choose `CC1[15]`, `CC1[14]` and `CC1[13]` to be bytes such that we get `PP2[15]=0x04`, `PP2[14]=0x04` and `PP2[13]=0x04` (because now we want to find fourth last byte and last 4 bytes in plaintext should be padded with `0x04`). We find `CC1[15] = D2[15]` $\oplus$ `0x04 = 0xce` $\oplus$ `0x04 = 0xca`. We find `CC1[14] = D2[14]` $\oplus$ `0x04 = 0x3b` $\oplus$ `0x04 = 0x3f`. We find `CC1[13] = D2[13]` $\oplus$ `0x04 = 0xf1` $\oplus$ `0x04 = 0xf5`. We set `CC1[15] = 0xca`, `CC1[14] = 0x3f` and `CC1[13] = 0xf5` in the attack program, `K=4` and run again. We find `CC1[12] = 0x18`. We find `D2[12] = CC1[12]` $\oplus$ `PP2[12] = 0x18` $\oplus$ `0x04 = 0x1c` (because `PP2[12] = 0x04`). We can compute the byte at 12 index of actual plaintext by `P2[12] = D2[12]` $\oplus$ `C1[12] = 0x1c` $\oplus$ `0xf2 = 0xee`.

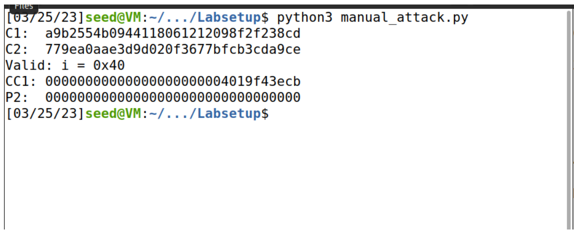


```
[03/25/23]seed@VM:~/.../Labsetup$ python3 manual_attack.py
C1:  a9b2554b0944118061212098f2f238cd
C2:  779ea0aae3d9d020f3677bfcb3cda9ce
Valid: i = 0x18
CC1: 0000000000000000000000018f53fca
P2:  00000000000000000000000000000000
[03/25/23]seed@VM:~/.../Labsetup$ 
```

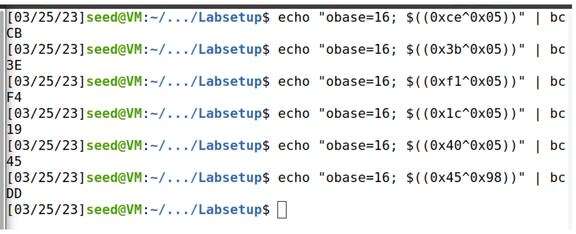


```
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xce^0x04))" | bc
CA
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x3b^0x04))" | bc
3F
[03/24/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xf1^0x04))" | bc
F5
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x18^0x04))" | bc
1C
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x1c^0xf2))" | bc
EE
[03/25/23]seed@VM:~/.../Labsetup$ 
```

We need to find `D2[11]`. Hence, we now need to construct another `CC1` that will give `D2[11]`. We choose `CC1[15]`, `CC1[14]`, `CC1[13]` and `CC1[12]` to be bytes such that we get `PP2[15]=0x05`, `PP2[14]=0x05`, `PP2[13]=0x05` and `PP2[12]=0x05` (because now we want to find fifth last byte and last 5 bytes in plaintext should be padded with `0x05`). We find `CC1[15] = D2[15]` $\oplus$ `0x05 = 0xce` $\oplus$ `0x05 = 0xcb`. We find `CC1[14] = D2[14]` $\oplus$ `0x05 = 0x3b` $\oplus$ `0x05 = 0x3e`. We find `CC1[13] = D2[13]` $\oplus$ `0x05 = 0xf1` $\oplus$ `0x05 = 0xf4`. We find `CC1[12] = D2[12]` $\oplus$ `0x05 = 0x1c` $\oplus$ `0x05 = 0x19`. We set `CC1[15] = 0xcb`, `CC1[14] = 0x3e`, `CC1[13] = 0xf4` and `CC1[12] = 0x19` in the attack program, `K=5` and run again. We find `CC1[11] = 0x40`. We find `D2[11] = CC1[11]` $\oplus$ `PP2[11] = 0x40` $\oplus$ `0x05 = 0x45` (because `PP2[11] = 0x05`). We can compute the byte at 11 index of actual plaintext by `P2[11] = D2[11]` $\oplus$ `C1[11] = 0x45` $\oplus$ `0x98 = 0xdd`.



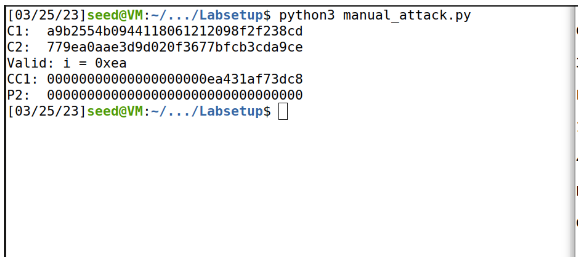
```
[03/25/23]seed@VM:~/.../Labsetup$ python3 manual_attack.py
C1:  a9b2554b0944118061212098f2f238cd
C2:  779ea0aae3d9d020f3677bfcb3cda9ce
Valid: i = 0x40
CC1: 000000000000000000004019f43ecb
P2:  00000000000000000000000000000000
[03/25/23]seed@VM:~/.../Labsetup$
```



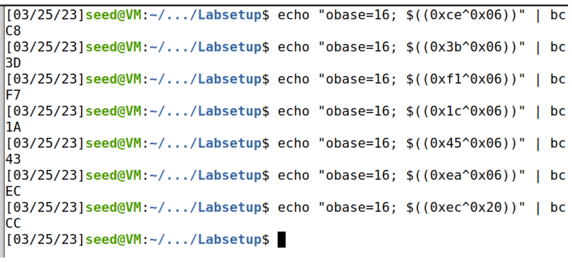
```
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xce^0x05))" | bc
CB
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x3b^0x05))" | bc
3E
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xf1^0x05))" | bc
F4
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x1c^0x05))" | bc
19
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x40^0x05))" | bc
45
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x45^0x98))" | bc
DD
[03/25/23]seed@VM:~/.../Labsetup$ 
```

We need to find `D2[10]`. Hence, we now need to construct another `CC1` that will give `D2[10]`. We choose `CC1[15]`, `CC1[14]`, `CC1[13]`, `CC1[12]` and `CC1[11]` to be bytes such that we get `PP2[15]=0x06`, `PP2[14]=0x06`, `PP2[13]=0x06`, `PP2[12]=0x06` and `PP2[11]=0x06` (because now we want to find sixth last byte and last 6 bytes in plaintext should be padded with `0x06`). We find `CC1[15] = D2[15]` $\oplus$ `0x06 = 0xce` $\oplus$ `0x06 = 0xc8`. We find `CC1[14] = D2[14]` $\oplus$ `0x06 = 0x3b` $\oplus$ `0x06 = 0x3d`. We find `CC1[13] = D2[13]` $\oplus$ `0x06 = 0xf1` $\oplus$ `0x06 = 0xf7`. We find `CC1[12] = D2[12]` $\oplus$ `0x06 = 0x1c` $\oplus$ `0x06 = 0x1a`. We find `CC1[11] = D2[11]` $\oplus$ `0x06 = 0x45` $\oplus$ `0x06 = 0x43`. We set `CC1[15] = 0xc8`, `CC1[14] = 0x3d`, `CC1[13]`

= `0xf7`, `CC1[13] = 0x1a` and `CC1[11] = 0x43` in the attack program, `K=6` and run again. We find `CC1[10] = 0xea`. We find `D2[10] = CC1[10]` ⊕ `PP2[10] = 0xea` ⊕ `0x06 = 0xec` (because `PP2[10] = 0x06`). We can compute the byte at 10 index of actual plaintext by `P2[10] = D2[10]` ⊕ `C1[10] = 0xec` ⊕ `0x20 = 0xcc`.



```
[03/25/23]seed@VM:~/.../Labsetup$ python3 manual_attack.py
C1:  a9b2554b0944118061212098f2f238cd
C2:  779ea0aae3d9d020f3677bfcb3cda9ce
Valid: i = 0xea
CC1: 00000000000000000000ea431af73dc8
P2:  00000000000000000000000000000000
[03/25/23]seed@VM:~/.../Labsetup$
```



```
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xce^0x06))" | bc
C8
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x3b^0x06))" | bc
3D
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xf1^0x06))" | bc
F7
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x1c^0x06))" | bc
1A
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0x45^0x06))" | bc
43
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xea^0x06))" | bc
EC
[03/25/23]seed@VM:~/.../Labsetup$ echo "obase=16; $((0xec^0x20))" | bc
CC
[03/25/23]seed@VM:~/.../Labsetup$
```

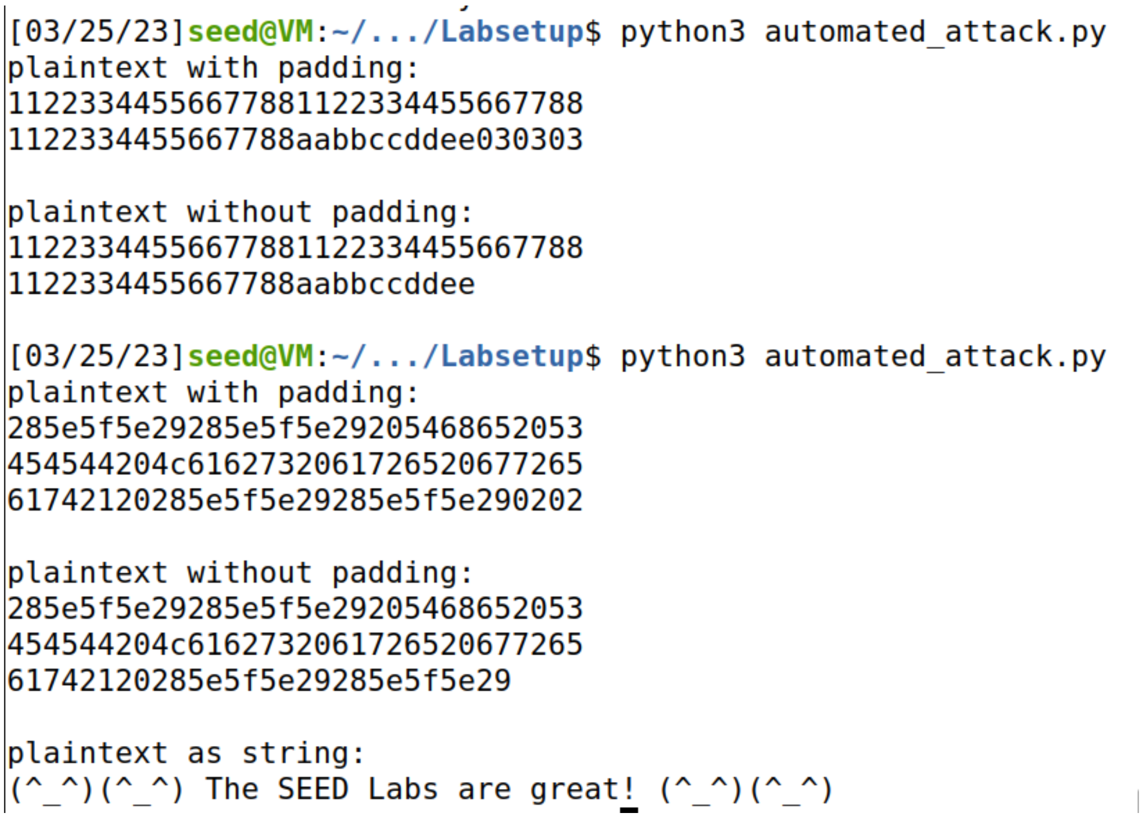
Hence, we have constructed the following byte arrays:

`D2[10:16] = [0xec, 0x45, 0x1c, 0xf1, 0x3b, 0xce]`

`P2[10:16] = [0xcc, 0xdd, 0xee, 0x03, 0x03, 0x03]`

4. **Task 3: Padding Oracle Attack (Level 2):**

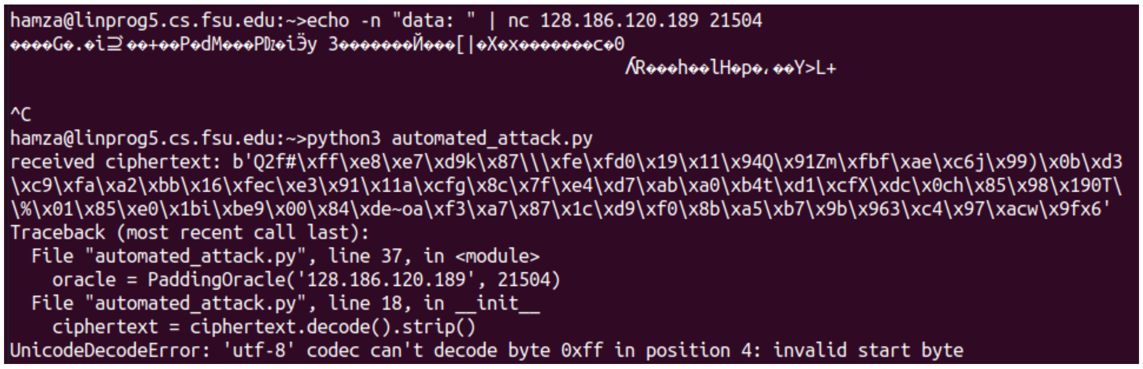
   I automated the attack done in previous task. The code is in `automated_attack.py`. To test, I ran the attack on previous oracle to print its secret message. It prints as expected, as shown below. The second run is for the actual oracle. For both runs, I printed the plaintext with padding and without. Also, I tried printing the plaintext as string for the second oracle and it prints a hidden message.



```
[03/25/23]seed@VM:~/.../Labsetup$ python3 automated_attack.py
plaintext with padding:
11223344556677881122334455667788
1122334455667788aabbccddee030303

plaintext without padding:
11223344556677881122334455667788
1122334455667788aabbccddee

[03/25/23]seed@VM:~/.../Labsetup$ python3 automated_attack.py
plaintext with padding:
285e5f5e29285e5f5e29205468652053
454544204c6162732061726520677265
61742120285e5f5e29285e5f5e290202

plaintext without padding:
285e5f5e29285e5f5e29205468652053
454544204c6162732061726520677265
61742120285e5f5e29285e5f5e29

plaintext as string:
(^_^)(^_^) The SEED Labs are great! (^_^)(^_^)
```

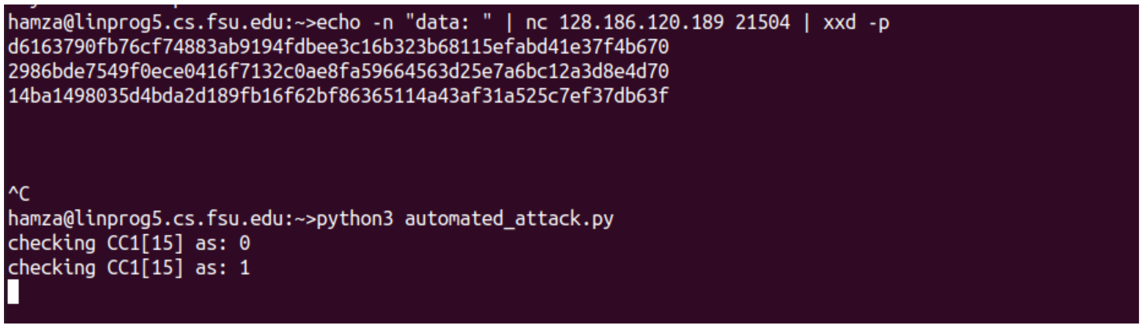
5. **Task 4: Demonstration:** I have tried the attack on the oracle present on linprog5.cs.fsu.edu

at port 21504 and 21505. Check program `automated_attack2.py`. First, I tried to automate the whole process, i.e., inside the `automated_attack2.py`, I get the ciphertext from the port 21504 and run the algorithm to reveal the secret message. However, what I get from the server as ciphertext does not seem like bytes, i.e., I am not able to convert it to hex bytes. Below, see the interaction through the `nc` command on shell, where it prints non-printable characters not recognizable. Then, I ran the automated attack and I get an error saying that the bytes cannot be decoded.



```
hamza@linprog5.cs.fsu.edu:~>echo -n "data: " | nc 128.186.120.189 21504
����G�.�i⊒��+��P�dM���PD̈�iЗy 3�������Й���[|�X�x��������c�0
                                                          ЃR���h��lH�p�,��Y>L+


^C
hamza@linprog5.cs.fsu.edu:~>python3 automated_attack.py
received ciphertext: b'Q2f#\xff\xe8\xe7\xd9k\x87\\\xfe\xfd0\x19\x11\x94Q\x91Zm\xfbf\xae\xc6j\x99)\x0b\xd3
\xc9\xfa\xa2\xbb\x16\xfec\xe3\x91\x11a\xcfg\x8c\x7f\xe4\xd7\xab\xa0\xb4t\xd1\xcfX\xdc\x0ch\x85\x98\x190T\
\%\x01\x85\xe0\x1bi\xbe9\x00\x84\xde~oa\xf3\xa7\x87\x1c\xd9\xf0\x8b\xa5\xb7\x9b\x963\xc4\x97\xacw\x9fx6'
Traceback (most recent call last):
  File "automated_attack.py", line 37, in <module>
    oracle = PaddingOracle('128.186.120.189', 21504)
  File "automated_attack.py", line 18, in __init__
    ciphertext = ciphertext.decode().strip()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 4: invalid start byte
```

I tried a less automated way, i.e., get the ciphertext from 21504, store it in the program, and the rest of the process is automated. However, as I am trying to figure out the CC1[15] by going through all 256 possibilities, I get stuck on second step. With CC1[15] as 0, I get the result "Invalid", I continue but then the server does not return any result. For now, I cannot reason why. I will try to work on it later some time, if time allows.



```
hamza@linprog5.cs.fsu.edu:~>echo -n "data: " | nc 128.186.120.189 21504 | xxd -p
d6163790fb76cf74883ab9194fdbee3c16b323b68115efabd41e37f4b670
2986bde7549f0ece0416f7132c0ae8fa59664563d25e7a6bc12a3d8e4d70
14ba1498035d4bda2d189fb16f62bf86365114a43af31a525c7ef37db63f



^C
hamza@linprog5.cs.fsu.edu:~>python3 automated_attack.py
checking CC1[15] as: 0
checking CC1[15] as: 1
█
```

References

[1] *The Padding Oracle Attack*. URL: `https://robertheaton.com/2013/07/29/padding-oracle-attack/`.