

**Faculty of Engineering and Technology Department of Electrical and Computer Engineering**

**ENCS4320 APPLIED CRYPTOGRAPHY**

**Padding Oracle Attack Lab**

**Prepared by: Nour Rabee’ ID:1191035**

**Instructor: Dr. Mohammed Hussein Section 2**

**Date of Submission: 2nd of July 2023**

**Abstract**

The Padding Oracle Attack Lab provides a hands-on opportunity to gain practical experience in exploiting vulnerabilities in cryptographic systems. In the lab, we work with two Oracle servers running inside a container, each containing a secret message encrypted with a secret key. By manipulating ciphertext and analyzing the responses received when checking padding validity, we aim to uncover the content of the secret messages. This lab enhances our understanding of the padding oracle attack, cryptographic vulnerabilities, and our ability to secure cryptographic implementations effectively

**Table of Contents**

[Table of Figures 4](#_Toc139213630)

[1. Theory 5](#_Toc139213631)

[What is Padding Oracle Attack? 5](#_Toc139213632)

[Padding scheme (PKCS5 Padding) for block ciphers 5](#_Toc139213633)

[2. Procedure 7](#_Toc139213634)

[2.1 Task 1: Getting Familiar with Padding 7](#_Toc139213635)

[2.2 Task 2: Padding Oracle Attack (Level 1) 9](#_Toc139213636)

[2.2.1 The Oracle Setup 9](#_Toc139213637)

[2.2.2 Deriving the Plaintext Manually 9](#_Toc139213638)

[2.3 Task 3: Padding Oracle Attack (Level 2) 15](#_Toc139213639)

[3. Conclusion 23](#_Toc139213640)

[4. References 24](#_Toc139213641)

# Table of Figures

Figure 1(PKCS#5 Padding) 6

Figure 2(5bytes File Padding) 7

Figure 3(10Bytes File Padding) 8

Figure 4(16Bytes File Padding) 8

Figure 5(interacting with the padding oracle hosted on IP address 10.9.0.80 and port 5000) 9

Figure 6(Results of deriving the plaintext manually) 10

Figure 7(Decryption using CBC) 11

Figure 8(interacting with the padding oracle hosted on IP address 10.9.0.80 and port 6000) 15

Figure 9(results of automation the Oracle attack at port 5000) 16

Figure 10(results of automation the Oracle attack at port 6000) 16

# 1. Theory

## What is Padding Oracle Attack?

A padding oracle attack is a type of attack against encrypted data that allows the attacker to decrypt the contents of the data, without knowing the key.

An oracle refers to a "tell" which gives an attacker information about whether the action they're executing is correct or not. Imagine playing a board or card game with a child. When their face lights up with a big smile because they think they're about to make a good move, that's an oracle. You, as the opponent, can use this oracle to plan your next move appropriately.

Padding is a specific cryptographic term. Some ciphers, which are the algorithms used to encrypt your data, work on blocks of data where each block is a fixed size. If the data you want to encrypt isn't the right size to fill the blocks, your data is padded until it does. Many forms of padding require that padding to always be present, even if the original input was of the right size. This allows the padding to always be safely removed upon decryption.

Putting the two things together, a software implementation with a padding oracle reveals whether decrypted data has valid padding. The oracle could be something as simple as returning a value that says "Invalid padding" or something more complicated like taking a measurably different time to process a valid block as opposed to an invalid block.

Block-based ciphers have another property, called the mode, which determines the relationship of data in the first block to the data in the second block, and so on. One of the most commonly used modes is CBC. CBC introduces an initial random block, known as the Initialization Vector (IV), and combines the previous block with the result of static encryption to make it such that encrypting the same message with the same key doesn't always produce the same encrypted output.

An attacker can use a padding oracle, in combination with how CBC data is structured, to send slightly changed messages to the code that exposes the oracle, and keep sending data until the oracle tells them the data is correct. From this response, the attacker can decrypt the message byte by byte.[[1]](https://learn.microsoft.com/en-us/dotnet/standard/security/vulnerabilities-cbc-mode)

## Padding scheme (PKCS5 Padding) for block ciphers

PKCS5 padding (Public-Key Cryptography Standards #5 padding) is a padding scheme used in cryptographic systems, particularly in block cipher modes of operation such as CBC (Cipher Block Chaining). It is used to ensure that the plaintext message being encrypted is properly aligned with the block size of the underlying block cipher.

To perform encryption with a block cipher in ECB or CBC mode the length of the input to be encrypted must be an exact multiple of the block length B in bytes. For Triple DES the block length B is 8 bytes (64 bits) and for all AES variants it is 16 bytes (128 bits). The most popular is "PKCS5" padding.

If the block length is B then add N padding bytes of value N to make the input length up to the next exact multiple of B. If the input length is already an exact multiple of B then add B bytes of value B. Thus padding of length N between one and B bytes is always added in an unambiguous manner. After decrypting, check that the last N bytes of the decrypted data all have value N with 1 < N ≤ B. If so, strip N bytes, otherwise throw a decryption error.

Examples of PKCS5 padding for block length B = 8:

1. 3 bytes: FDFDFD

Padding process:

* The input length is not a multiple of the block size (3 is not divisible by 8), so padding is needed.
* We need to add 5 bytes to reach the next exact multiple of the block size.
* PKCS5 padding adds padding bytes with a value equal to the number of padding bytes added.
* Padding bytes with a value of 5 (0x05) are added.

After adding the padding, the padded input becomes FDFDFD0505050505.

1. 7 bytes: FDFDFDFDFDFDFD

Padding process:

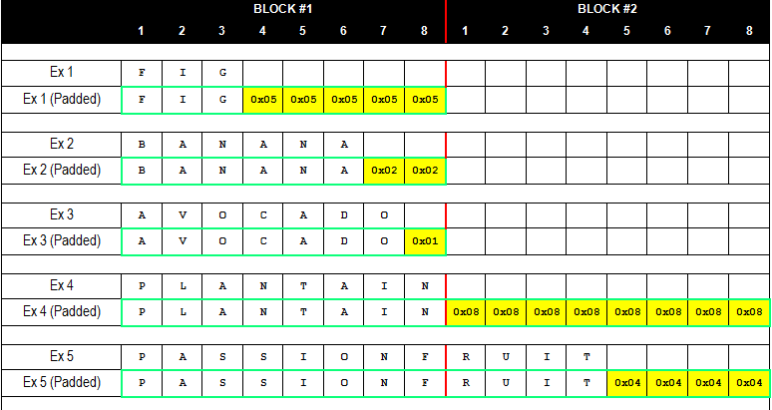
* + The input length is not a multiple of the block size (7 is not divisible by 8), so padding is needed.
  + We need to add 1 byte to reach the next exact multiple of the block size.
  + PKCS5 padding adds padding bytes with a value equal to the number of padding bytes added.
  + Padding bytes with a value of 1 (0x01) are added.

After adding the padding, the padded input becomes FDFDFDFDFDFDFD01

1. 8 bytes: FDFDFDFDFDFDFDFD

Padding process:

* + The input length is already a multiple of the block size (8 bytes), but in order to indicate padding, we can add a full block of padding.
  + Since the blocksize is 8 bytes, we add 8 bytes of padding, with each padding byte having a value of 8 (0x08).

After adding the padding, the padded input becomes FDFDFDFDFDFDFDFD0808080808080808.

*Figure 1(PKCS#5 Padding)*

# Procedure

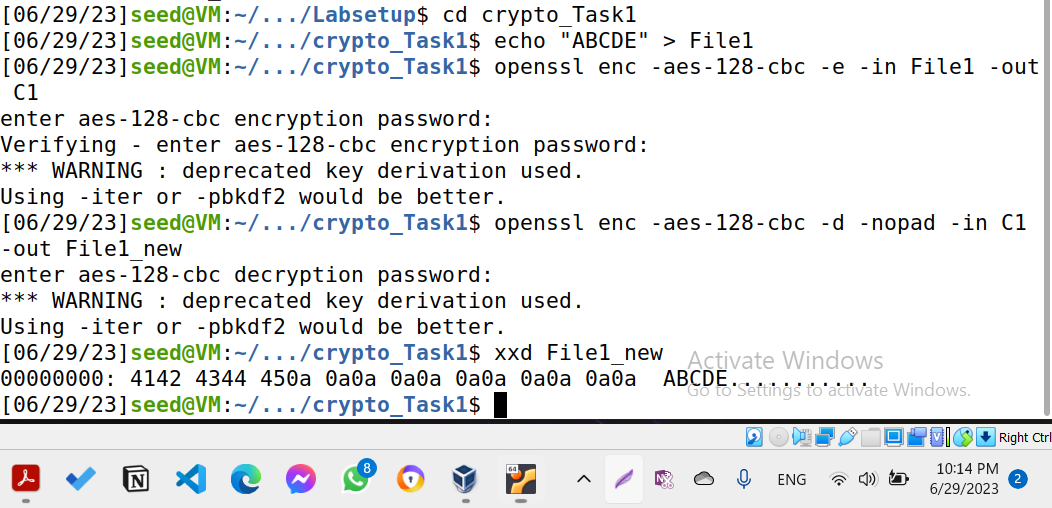
### Task 1: Getting Familiar with Padding

Three files have been created, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. Some commands have been used to figure out what paddings are added to them.

We created a directory called cryto\_Task1 using the command (mkdir).

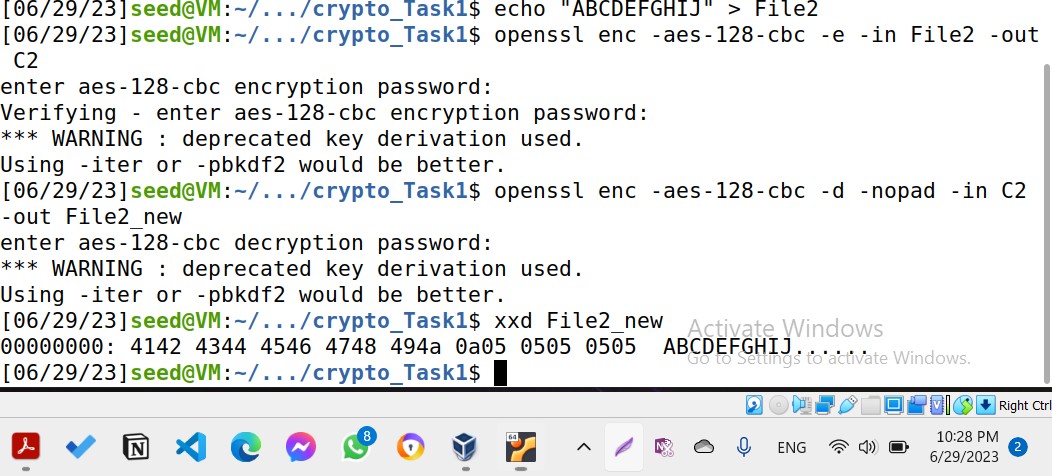
The specified steps were carried out on a 5-byte file, and the same set of steps was also applied to both the 10-byte and 16-byte files.

1. “$ echo -n "ABCDE" > File1” this command is used to write the string "ABCDE" to a file named "File1" without including a trailing newline character.
2. “$ openssl enc -aes-128-cbc -e -in File1 -out C1” this command utilizes the OpenSSL tool to encrypt the contents of "File1" using the AES-128-CBC encryption algorithm. The resulting encrypted data “Ciphertext” is stored in a file named "C1"
3. $ openssl enc -aes-128-cbc -d -nopad -in C1 -out File1\_new This command employs the OpenSSL tool to decrypt the ciphertext in the file "C1" using the AES-128-CBC decryption algorithm. The resulting plaintext is stored in a new file named "File1\_new". The "-nopad" option indicates that no padding is expected in the ciphertext.
4. “$ xxd File1\_new” This command is used to display the hexadecimal representation of the contents of the file "File1\_new".

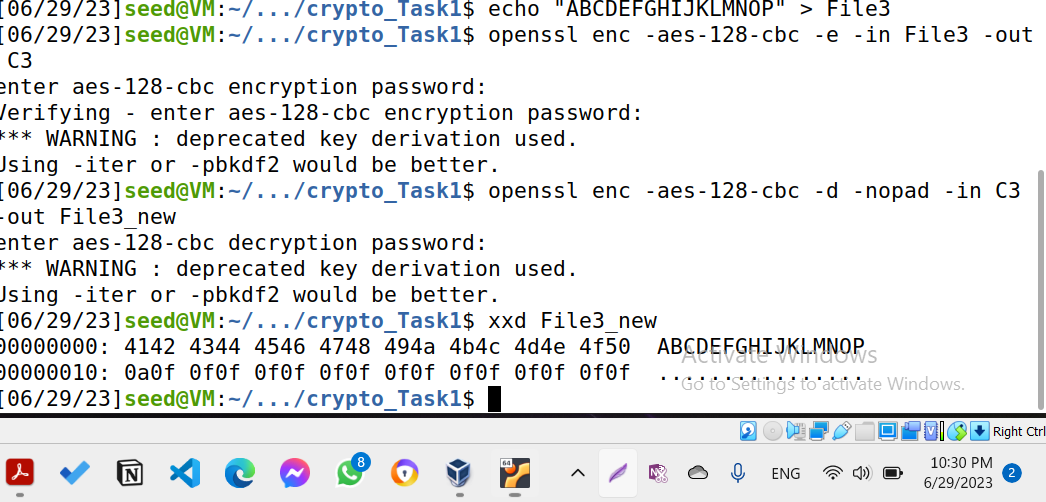


*Figure 2(5bytes File Padding)*

We realized that “0a” has been repeated 11 times before the original plaintext.



*Figure 3(10Bytes File Padding)*



*Figure 4(16Bytes File Padding)*

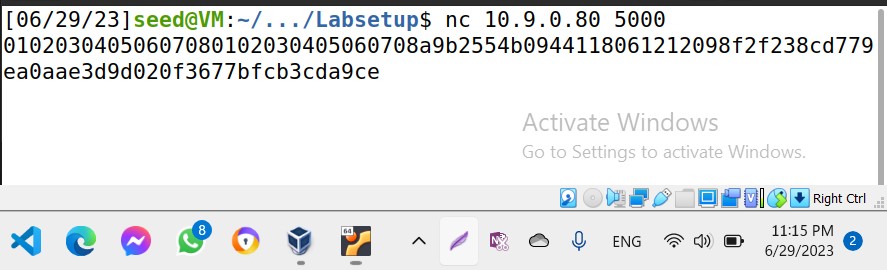
The same steps were performed for the remaining files, one with 10 bytes and another with 16 bytes. The results for each file were observed and displayed as shown in Figure 3 and Figure 4, respectively.

### Task 2: Padding Oracle Attack (Level 1)

## The Oracle Setup

In this task, there is a padding oracle running on port 5000 that uses the AES-CBC encryption algorithm and mode. The secret message is encrypted with an unknown key called K. You can communicate with the oracle by connecting to it using the command "nc 10.9.0.80 5000".

When you establish a connection with the oracle, it will provide you with hexadecimal data. The first 16 bytes represent the Initialization Vector (IV), and the remaining bytes represent the ciphertext. The ciphertext is 32 bytes long, which corresponds to two blocks. However, the length of the original plaintext is unknown due to the presence of padding.



*Figure 5(interacting with the padding oracle hosted on IP address 10.9.0.80 and port 5000*)

## Deriving the Plaintext Manually

The objective is to determine the plaintext of a secret message encrypted using AES-128-CBC with an unknown key. The ciphertext consists of two blocks, P1 and P2, and the focus is on retrieving the second block, P2.

The provided skeleton code in the lab manual, initializes the D2 and CC1 arrays. In each iteration, we focus on one byte of CC1 and try all 256 possible values for that byte. we construct the ciphertext CC1 + C2 (along with the IV) and send it to the oracle. If the padding is valid, it means the value you tried for that byte of CC1 is correct, allowing you to determine one byte of D2.

By manually adjusting the value of K, we can utilize the execution results in each iteration to update D2 accordingly. Afterward, we can rerun the program with an updated CC1 to obtain the next byte of D2, for instance D2[14]. Continuing this process, you can acquire D2[13], D2[12], and so on, until all the bytes of D2 are obtained. Once the complete D2 is obtained, we will have the plaintext value of P2.

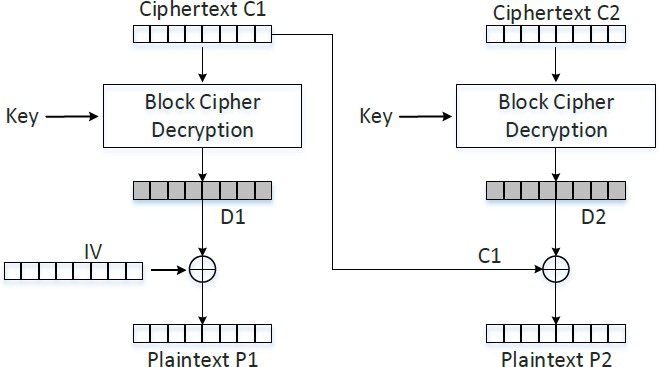
For the sake of simplicity, we only have to return the first 6 bytes if p2 manually.



*Figure 6(Results of deriving the plaintext manually)*

As mentioned, when establishing a connection with the oracle using the command "nc 10.9.0.80 5000," we receive 32 bytes of ciphertext, excluding the 16-byte IV. In the manual, the plaintext is provided in hexadecimal format for debugging purposes. The length of the plaintext is 29 bytes.

The ciphertext is converted back to plaintext during the decryption process, including the padding. From figure 6, it is apparent that three padding bytes with a value of 3 (0x03) were added to the plaintext. This padding ensures that the plaintext reaches the next exact multiple of the next block size, which is 16 bytes in this case to reach 32 bytes.



*Figure 7(Decryption using CBC)*

Since C2 remains unchanged that means that P2 remains unchanged as well, but we do not know its value yet. So, we are going to modify the value of C1 (CC1  the modified C1). For the first iteration, we are going to only change the last byte of CC1 and try all 256 possible values for that byte, since we are only trying to change the last byte of the plaintext (P2) to become 01**(PKCS5 Padding)**. So we only need a simple for loop looping through and searching through 0 to 255 for the last byte, and when the padding oracle finally outputs “valid”, then we will know what value CC1[15] needs to be. At this point of time we know CC1[15] and we also know P2[15] because we are forcing it to become 01. Therefore, because of the xor operation, we can simply xoring CC1[15] with 01 and retrieve the last byte in D2. In the second iteration, we are forcing the last two bytes of P2 to become 02. As we mentioned, since decryption is deterministic algorithm, if we do not change C2 this means that D2 will also remain the same, because of that we are not going to update the values in P2. So, CC1[15] becomes D [15] xor 02, CC1[14] is obtained in a similar manner as CC1[15] in the first iteration, and D2[14] will be the xoring of CC1[14] with

02. The same process will be done until all bytes of P2 have been recovered.

To summarize the process:

1. Initialize K = 1.
2. Iterate through all possible values (0 to 255) for the last byte of CC1 (CC1[15]):
   * Set CC1[15] = i.
   * Decrypt the modified ciphertext (IV + CC1 + C2) using the padding oracle.
   * If the padding is valid (oracle outputs "valid"), retrieve the last byte of D2 by xoring CC1[15] with 0x01.
3. Set K = 2.
4. Set CC1[15] = D2[15] XOR 0x02.
5. Iterate through all possible values (0 to 255) for the second-to-last byte of CC1 (CC1[14]):
   * Set CC1[14] = i.
   * Decrypt the modified ciphertext (IV + CC1 + C2) using the padding oracle.
   * If the padding is valid (oracle outputs "valid"), retrieve the second-to-last byte of D2 by xoring CC1[14] with 0x02.
6. Repeat steps 3-5 for each byte in CC1 and D2, moving from the second-to-last byte to the first byte, updating K and CC1 accordingly.
7. Once you have determined all the bytes in D2, you have successfully recovered the second block of the plaintext P2.

It's important to note that in each iteration, we are modifying CC1 to force specific padding values in the decrypted plaintext. By observing the padding oracle's response, we can determine the correct values for each byte in D2 and eventually recover the entire second block of the plaintext P2.

All the previous steps have done manually using the code below.

#!/usr/bin/python3 import socket

from binascii import hexlify, unhexlify

# XOR two bytearrays def xor(first, second):

return bytearray(x^y for x,y in zip(first, second)) class PaddingOracle:

def init (self, host, port) -> None:

self.s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) self.s.connect((host, port))

ciphertext = self.s.recv(4096).decode().strip() self.ctext = unhexlify(ciphertext)

def decrypt(self, ctext: bytes) -> None: self.\_send(hexlify(ctext))

return self.\_recv()

def \_recv(self):

resp = self.s.recv(4096).decode().strip() return resp

def \_send(self, hexstr: bytes): self.s.send(hexstr + b'\n')

def del (self): self.s.close()

if name == " main ":

oracle = PaddingOracle('10.9.0.80', 5000)

# Get the IV + Ciphertext from the oracle iv\_and\_ctext = bytearray(oracle.ctext)

IV = iv\_and\_ctext[00:16]

C1 = iv\_and\_ctext[16:32] # 1st block of ciphertext C2 = iv\_and\_ctext[32:48] # 2nd block of ciphertext print("C1: " + C1.hex())

print("C2: " + C2.hex())

###############################################################

# Here, we initialize D2 with C1, so when they are XOR-ed, # The result is 0. This is not required for the attack.

# Its sole purpose is to make the printout look neat.

# In the experiment, we will iteratively replace these values. D2 = bytearray(16)

D2[0] = C1[0]

D2[1] = C1[1]

D2[2] = C1[2]

D2[3] = C1[3]

D2[4] = C1[4]

D2[5] = C1[5]

D2[6] = C1[6]

D2[7] = C1[7]

D2[8] = C1[8]

D2[9] = C1[9] D2[10] = 0xec D2[11] = 0x45 D2[12] = 0x1c D2[13] = 0xf1 D2[14] = 0x3b

D2[15] = 0xce #CC1[15] xor K=1 ###############################################################

# In the experiment, we need to iteratively modify CC1

# We will send this CC1 to the oracle, and see its response. CC1 = bytearray(16)

CC1[0] = 0x00 CC1[1] = 0x00 CC1[2] = 0x00 CC1[3] = 0x00

CC1[4] = 0x00 CC1[5] = 0x00 CC1[6] = 0x00 CC1[7] = 0x00 CC1[8] = 0x00 CC1[9] = 0x00 CC1[10] = 0xeb CC1[11] = 0x42 CC1[12] = 0x1b CC1[13] = 0xf6 CC1[14] = 0x3c

CC1[15] = 0xc9 #D2[] xor k

###############################################################

# In each iteration, we focus on one byte of CC1.

# We will try all 256 possible values, and send the constructed # ciphertext CC1 + C2 (plus the IV) to the oracle, and see

# which value makes the padding valid.

# As long as our construction is correct, there will be

# one valid value. This value helps us get one byte of D2.

# Repeating the method for 16 times, we get all the 16 bytes of D2.

K = 7

for i in range(256):

CC1[16 - K] = i

status = oracle.decrypt(IV + CC1 + C2) if status == "Valid":

print("Valid: i = 0x{:02x}".format(i))

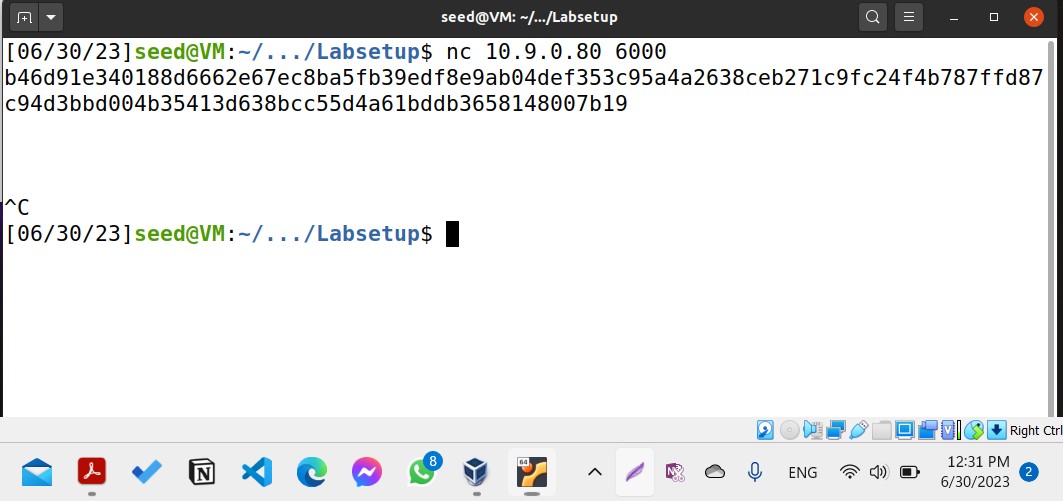
print("CC1: " + CC1.hex()) ###############################################################

# Once you get all the 16 bytes of D2, you can easily get P2 P2 = xor(C1, D2)

print("P2: " + P2.hex())

### Task 3: Padding Oracle Attack (Level 2)

In this task, we will automate the attack process, and this time, we need to get all the blocks of the plaintext.



*Figure 8(interacting with the padding oracle hosted on IP address 10.9.0.80 and port 6000)*

The command "nc 10.9.0.80 6000" establishes a network connection to the host at IP address

10.9.0.80 on port 6000 and retrieves the ciphertext. The ciphertext represents the encrypted form of a message or data. The provided ciphertext consists of multiple blocks, including IV, C1, C2, and C3, each consisting of 16 bytes based on AES-128-CBC. It’s obvious that the original plain text will be P1|| P2 ||P3. By following the same process as in task 2, we can decrypt and retrieve the bytes of each block (P1, P2, and P3) from the given ciphertext to recover all the plaintext.

For the sake of debugging, I have automated that arratck process for both ports 5000, and 6000.

The following code demonstrates how an Oracle padding attack can be performed when the plaintext consists of only two blocks. The attack is executed by establishing a connection with the oracle on port 5000 (task2). The same steps that have been discussed in the previous task have been done here also for both ports 5000 and 6000.

import socket

from binascii import hexlify, unhexlify

# XOR two bytearrays def xor(first, second):

return bytearray(x ^ y for x, y in zip(first, second))

class PaddingOracle:

def init (self, host, port) -> None:

self.s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) self.s.connect((host, port))

ciphertext = self.s.recv(4096).decode().strip() self.ctext = unhexlify(ciphertext)

def decrypt(self, ctext: bytes) -> None: self.\_send(hexlify(ctext))

return self.\_recv()

def \_recv(self):

resp = self.s.recv(4096).decode().strip() return resp

def \_send(self, hexstr: bytes): self.s.send(hexstr + b'\n')

def del (self): self.s.close()

if name == " main ":

oracle = PaddingOracle('10.9.0.80', 5000)

# Get the IV + Ciphertext from the oracle iv\_and\_ctext = bytearray(oracle.ctext) IV = iv\_and\_ctext[00:16]

C1 = iv\_and\_ctext[16:32] # 1st block of ciphertext C2 = iv\_and\_ctext[32:48] # 2nd block of ciphertext print("IV: " + IV.hex())

print("C1: " + C1.hex())

print("C2: " + C2.hex())

D2 = bytearray(16) CC1 = bytearray(16) D1 = bytearray(16)

modified\_iv = bytearray(16)

#Iterate through all bytes of CC1 and D2 for K in range(1, 17):

for i in range(256):

CC1[16 - K] = i

status = oracle.decrypt(IV + CC1 + C2) if status == "Valid":

print("Valid: i = 0x{:02x}".format(i))

print("CC1: " + CC1.hex()) D2[16 - K] = CC1[16 - K] ^ K

for j in range(16 - K, 16):

CC1[j] = D2[j] ^ (K + 1)

break

print(" ")

print("D2: " + D2.hex())

# Once you have determined all the bytes in D2, you have successfully recovered the second block of the plaintext P2.

P2 = xor(C1, D2)

print("P2: " + P2.hex())

# Iterate through all bytes of the modified IV and D1 for K in range(1, 17):

for i in range(256): modified\_iv[16 - K] = i

status = oracle.decrypt(IV + modified\_iv + C1) if status == "Valid":

print("Valid: i = 0x{:02x}".format(i)) print("modified\_iv: " + modified\_iv.hex())

D1[16 - K] = modified\_iv[16 - K] ^ K for j in range(16 - K, 16):

modified\_iv[j] = D1[j] ^ (K + 1) break

print("

print("D1: " + D2.hex())

")

# Once you have determined all the bytes in D1, you have successfully

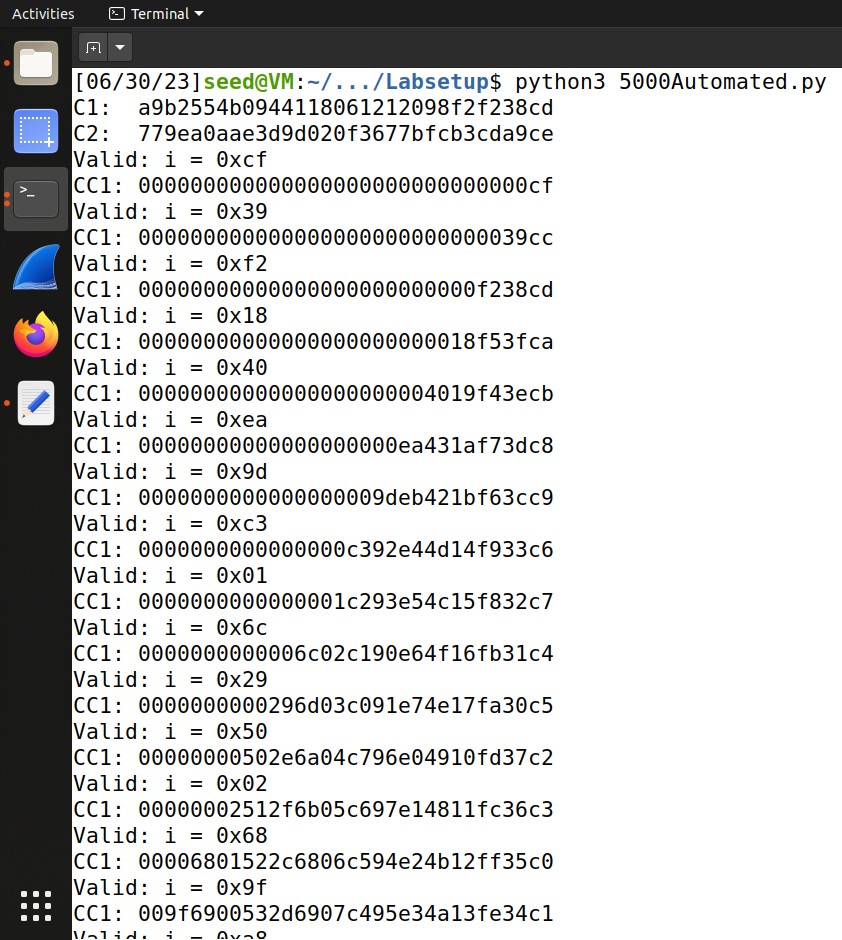
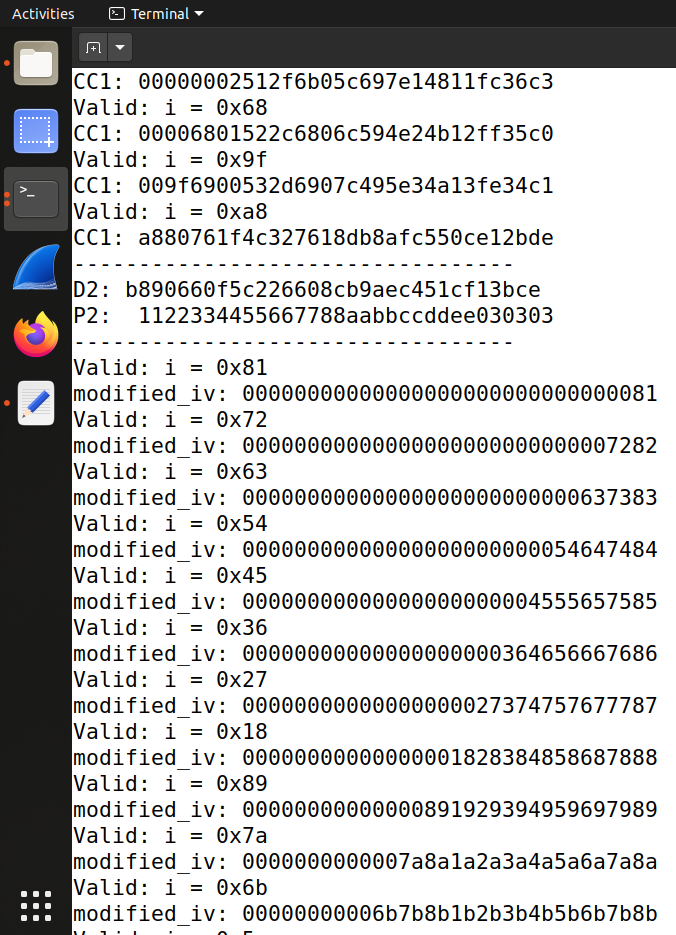
recovered the second block of the plaintext P1.

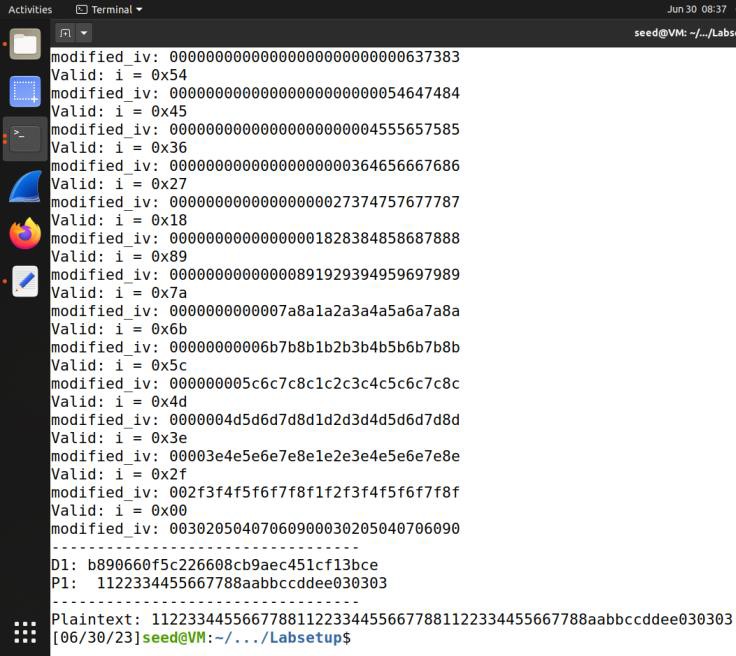
P1 = xor(IV, D1)

print("P1: " + P2.hex())

P = P1 + P2

print("PlainText: " + P.hex())



*Figure 9(results of automation the Oracle attack at port 5000)*

In the manual, the plaintext is provided in hexadecimal format for debugging purposes. The length of the plaintext is 29 bytes.

The ciphertext is converted back to plaintext during the decryption process, including the padding. From figure 9, it is apparent that three padding bytes with a value of 3 (0x03) were added to the plaintext. This padding ensures that the plaintext reaches the next exact multiple of the next block size, which is 16 bytes in this case to reach 32 bytes. (p.s our results exactly match the one in the manual).

The same will be done for port 6000, but with some changes. The connection with the Oracle at port 6000 return IV and 3 block ciphers which means that the plaintext consists of 3 blocks each 16 bytes as shown in figure 8.

So we will return P1, P2, and P3, concatenate them together to retrieve the plaintext added to it padding values.

#!/usr/bin/python3 import socket

from binascii import hexlify, unhexlify

# XOR two bytearrays def xor(first, second):

return bytearray(x^y for x,y in zip(first, second)) class PaddingOracle:

def init (self, host, port) -> None:

self.s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) self.s.connect((host, port))

ciphertext = self.s.recv(4096).decode().strip() self.ctext = unhexlify(ciphertext)

def decrypt(self, ctext: bytes) -> None: self.\_send(hexlify(ctext))

return self.\_recv()

def \_recv(self):

resp = self.s.recv(4096).decode().strip() return resp

def \_send(self, hexstr: bytes): self.s.send(hexstr + b'\n')

def del (self): self.s.close()

if name == " main ":

oracle = PaddingOracle('10.9.0.80', 6000)

# Get the IV + Ciphertext from the oracle

iv\_and\_ctext = bytearray(oracle.ctext)

IV = iv\_and\_ctext[00:16]

C1 = iv\_and\_ctext[16:32] # 1st block of ciphertext C2 = iv\_and\_ctext[32:48] # 2nd block of ciphertext C3 = iv\_and\_ctext[48:64]

print("IV: " + IV.hex())

print("C1: " + C1.hex())

print("C2: " + C2.hex())

print("C3: " + C3.hex())

###############################################################

# Here, we initialize D2 with C1, so when they are XOR-ed, # The result is 0. This is not required for the attack.

# Its sole purpose is to make the printout look neat.

# In the experiment, we will iteratively replace these values.

D1 = bytearray(16) D2 = bytearray(16) D3 = bytearray(16)

IV\_modefied = bytearray(16) CC1 = bytearray(16)

CC2 = bytearray(16)

###############################################################

# In each iteration, we focus on one byte of CC1.

# We will try all 256 possible values, and send the constructed # ciphertext CC1 + C2 (plus the IV) to the oracle, and see

# which value makes the padding valid.

# As long as our construction is correct, there will be

# one valid value. This value helps us get one byte of D2.

# Repeating the method for 16 times, we get all the 16 bytes of D2.

# For D3

for K in range(1 , 17): for i in range(256):

CC2[16 - K] = i

status = oracle.decrypt(IV + CC2 + C3) if status == "Valid":

print("Valid: i = 0x{:02x}".format(i))

print("CC2: " + CC2.hex()) D3[16 - K] = CC2[16 - K] ^ K

for j in range(16 - K ,16): CC2[j] = D3[j] ^ (K + 1)

break ###############################################################

# For D2

for K in range(1 , 17): for i in range(256):

CC1[16 - K] = i

status = oracle.decrypt(IV + CC1 + C2) if status == "Valid":

print("Valid: i = 0x{:02x}".format(i))

print("CC1: " + CC1.hex()) D2[16 - K] = CC1[16 - K] ^ K

for j in range(16 - K ,16): CC1[j] = D2[j] ^ (K + 1)

break ###############################################################

# For D1

for K in range(1 , 17): for i in range(256):

IV\_modefied[16 - K] = i

status = oracle.decrypt(IV + IV\_modefied+ C1) if status == "Valid":

print("Valid: i = 0x{:02x}".format(i)) print("IV\_modefied: " + IV\_modefied.hex())

D1[16 - K] = IV\_modefied[16 - K] ^ K for j in range(16 - K ,16):

IV\_modefied[j] = D1[j] ^ (K + 1) break

###############################################################

# Once you get all the 16 bytes of D2, you can easily get P2 P1 = xor(IV, D1)

P2 = xor(C1, D2) P3 = xor(C2, D3)

print(" ")

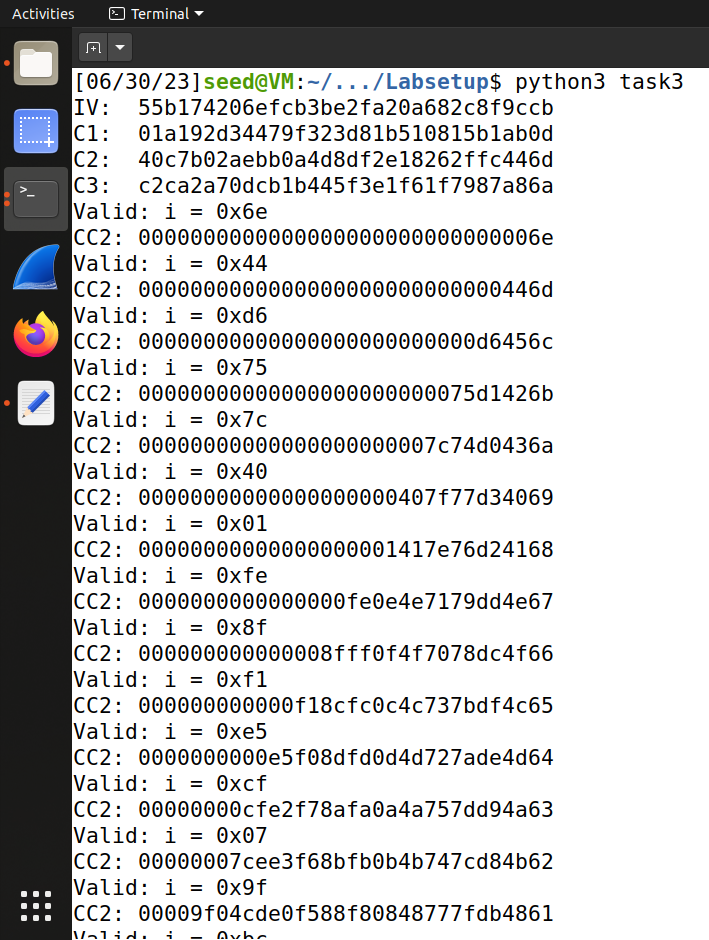
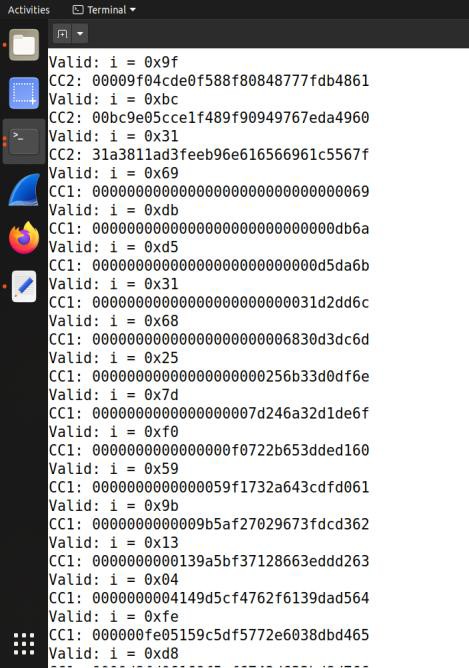
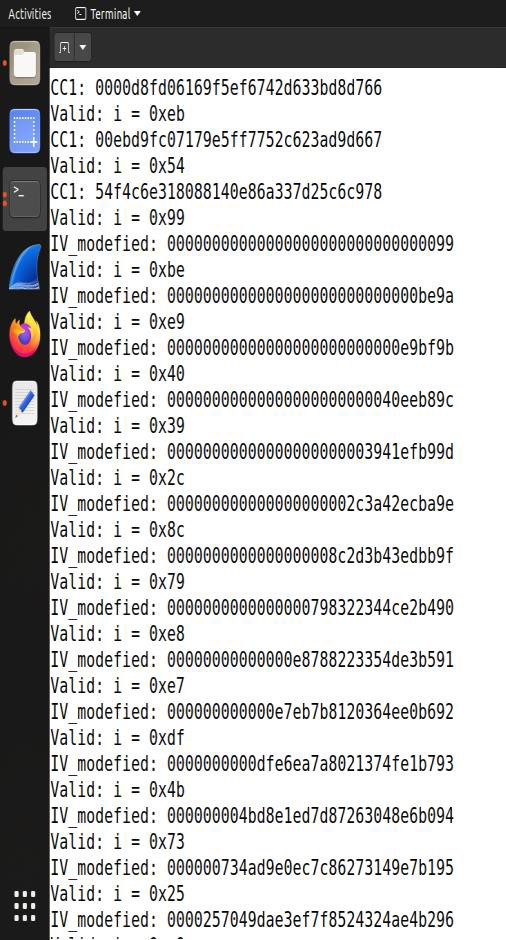
print("P1: " + P1.hex())

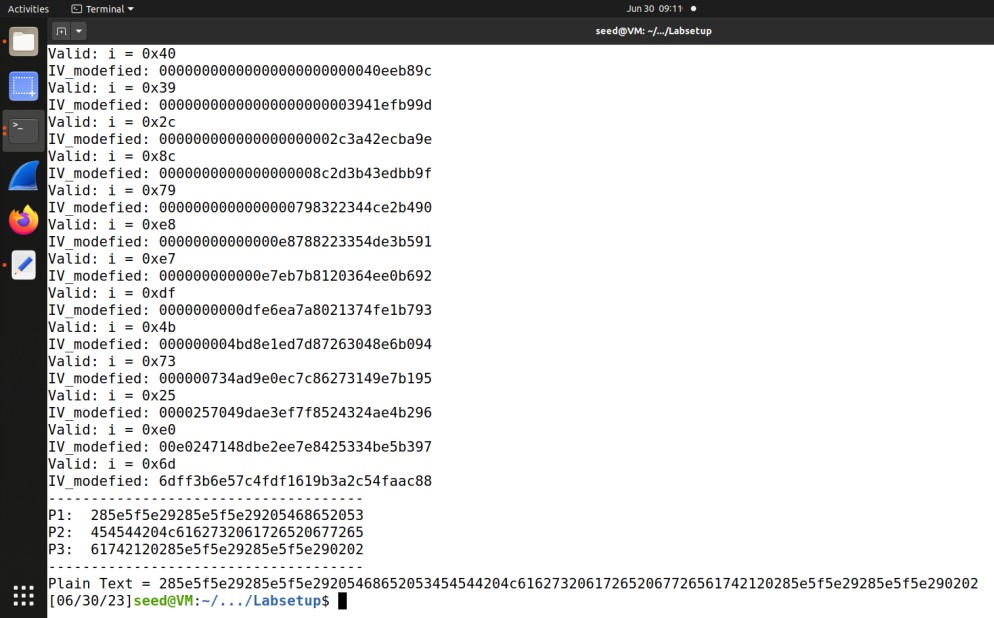
print("P2: " + P2.hex())

print("P3: " + P3.hex())

print(" ")

Plain\_Text = P1.hex() + P2.hex() + P3.hex() print("Plain Text = " + Plain\_Text)



*Figure 10(results of automation the Oracle attack at port 6000)*

The ciphertext is converted back to plaintext during the decryption process, including the padding. From figure 10, it is apparent that two padding bytes with a value of 2 (0x02) were added to the plaintext. This padding ensures that the plaintext reaches the next exact multiple of the next block size, which is 16 bytes in this case to reach 48 bytes.

# 3. Conclusion

To sum up, in this project, we successfully applied the padding oracle attack to decrypt ciphertext from two different ports, 5000 and 6000, and recover the corresponding plaintext. The attack involved manipulating the ciphertext and observing the server's responses to determine the correct values for each decrypted plaintext block. By iterating through the blocks and adjusting the ciphertext accordingly, we were able to exploit the vulnerability in the padding scheme and extract the plaintext. This project showcased the effectiveness of the padding oracle attack in revealing sensitive information and emphasized the importance of robust encryption practices to prevent such attacks.

# 4. References

[1] <https://learn.microsoft.com/en-us/dotnet/standard/security/vulnerabilities-cbc-mode>

[2]<https://www.cryptosys.net/pki/manpki/pki_paddingschemes.html#:~:text=PKCS5%20Padding,added%20in%20an%20unambiguous%20manner.>