



**FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING**

ENCS4320 - APPLIED CRYPTOGRAPHY

Second semester 2022/2023

Padding Oracle Attack Lab

Report

Prepared by:

Tariq Odeh – 1190699

Osama Qutait – 1191072

Sec: 2

Instructor: Dr. Mohammed Hussein

Date: 27th Jun 2023

Abstract

In this lab, a hands-on experience with a padding oracle attack will be observed and learned. Errors can be exposed by some systems when the padding is invalid during the decryption process of a given ciphertext. The lab provides two oracle servers within a container, each containing a secret message. Ciphertexts and the corresponding initialization vectors (IVs) can be sent to the oracles, which will decrypt the ciphertexts and indicate the validity of the padding. The objective is to leverage these responses to uncover the hidden secret messages. The lab covers key topics such as secret-key encryption, encryption modes, paddings, and the mechanics of the padding oracle attack. The lab will be tested on the SEED Ubuntu 20.04 VM, which can be obtained as a pre-built image from the SEED website.

Table of Contents

1. Theory	4
1.1. Secret Key Cryptography	4
1.2. Padding oracle attack	4
1.3. PKCS padding method	4
1.4. Cipher block chaining (CBC).....	5
1.5. Advanced Encryption Standard (AES)	5
2. Build Lab Environment	6
3. Task 1: Getting Familiar with Padding	7
3.1. File with 5 bytes	7
3.2. File with 10 bytes	8
3.3. File with 16 bytes	9
4. Task 2: Padding Oracle Attack (Level 1)	10
5. Task 3: Padding Oracle Attack (Level 2)	13
6. Conclusion	16
7. References	17
8. Appendix	18
8.1. Task 2 - Code	18
8.2. Task 3 - Code	20

List of Figures

Figure 1-1: Secret key cryptography	4
Figure 1-2: cipher block chaining (CBC) decryption [4]	5
Figure 2-1: Commonly used commands related to Docker and Compose	6
Figure 3-1: File with 5 bytes.....	7
Figure 3-2: File with 10 bytes	8
Figure 3-3: File with 16 bytes	9
Figure 4-1: Padding Oracle Attack (Level 1) code – 1	10
Figure 4-2: Padding Oracle Attack (Level 1) code – 2.....	11
Figure 4-3: Process to decrypt it entirely - 1	12
Figure 4-4: Process to decrypt it entirely - 2	12
Figure 5-1: Padding Oracle Attack (Level 2) -1.....	13
Figure 5-2: Padding Oracle Attack (Level 2) -2.....	13
Figure 5-3: Process to decrypt it entirely - 1	14
Figure 5-4: Process to decrypt it entirely - 2	14
Figure 5-5: Process to decrypt it entirely - 3	15
Figure 5-6: Process to decrypt it entirely - 4	15

1. Theory

1.1. Secret Key Cryptography

In secret-key cryptography, the same key is used by both parties (Alice and Bob) for encrypting and decrypting messages. Agreement on the cryptographic algorithm and key exchange is required. However, the logistical challenge arises when securely transferring the key to prevent access by an attacker. If the key is compromised, the attacker can decrypt intercepted messages and impersonate one of the parties. [1]



Figure 1-1: Secret key cryptography

1.2. Padding oracle attack

In cryptography, a padding oracle attack is an attack in which the padding validation of a cryptographic message is used to decrypt the ciphertext. In cryptography, the padding of variable-length plaintext messages, often required for compatibility with the underlying cryptographic primitive, is exploited by the attack. The attack is dependent on the presence of a "padding oracle" that freely responds to queries regarding the correctness of the padding in a message. CBC mode decryption employed in block ciphers is primarily associated with padding oracle attacks. Padding oracle attacks can also potentially exploit padding modes utilized in asymmetric algorithms such as OAEP. [2]

1.3. PKCS padding method

When utilizing the Symmetric Algorithm Encipher callable service, padding is implemented prior to encryption, whereas it is eliminated from decrypted data when employing the Symmetric Algorithm Decipher callable service. The rules governing PKCS padding are straightforward: padding bytes are consistently appended to the plaintext before encryption. Each padding byte holds a value equivalent to the total number of added padding bytes. For instance, if six padding bytes are required, each of those bytes will have a value of 0x06. The minimum number of padding bytes is always one, and it ensures that the data length becomes a multiple of the cipher algorithm block size. [3]

1.4. Cipher block chaining (CBC)

Cipher block chaining (CBC) is a block cipher mode of operation that encrypts a sequence of bits as a single block using a cipher key. It employs an initialization vector (IV) and ensures secure encryption and decryption of large amounts of plaintext. CBC's chaining process makes the decryption of a ciphertext block dependent on all preceding blocks, with the integrity of previous blocks contained in the adjacent ciphertext block. Any error in a ciphertext block affects the decryption of subsequent blocks, and rearranging the ciphertext block order can corrupt the decryption process. [4]

- Encryption

$$C_i = E_k(P_i \oplus C_{i-1}), \text{ and } C_0 = IV \quad 1.1$$

- Decryption

$$P_i = D_k(C_i) \oplus C_{i-1}, \text{ and } C_0 = IV \quad 1.2$$

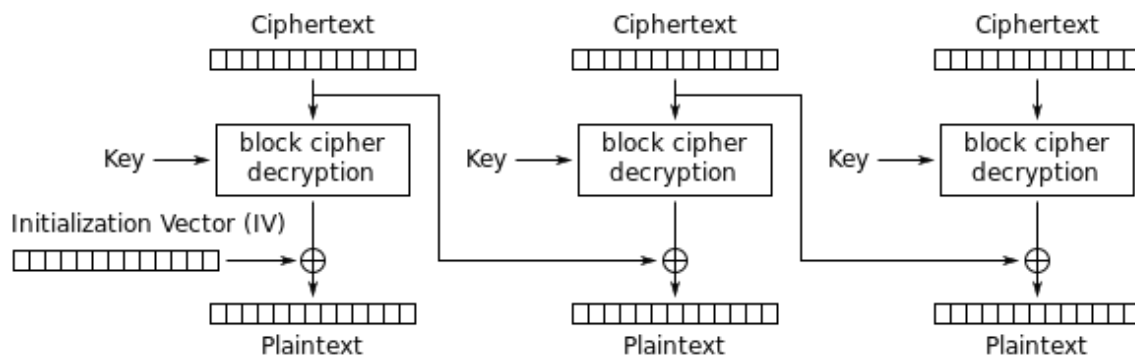


Figure 1-2: cipher block chaining (CBC) decryption [4]

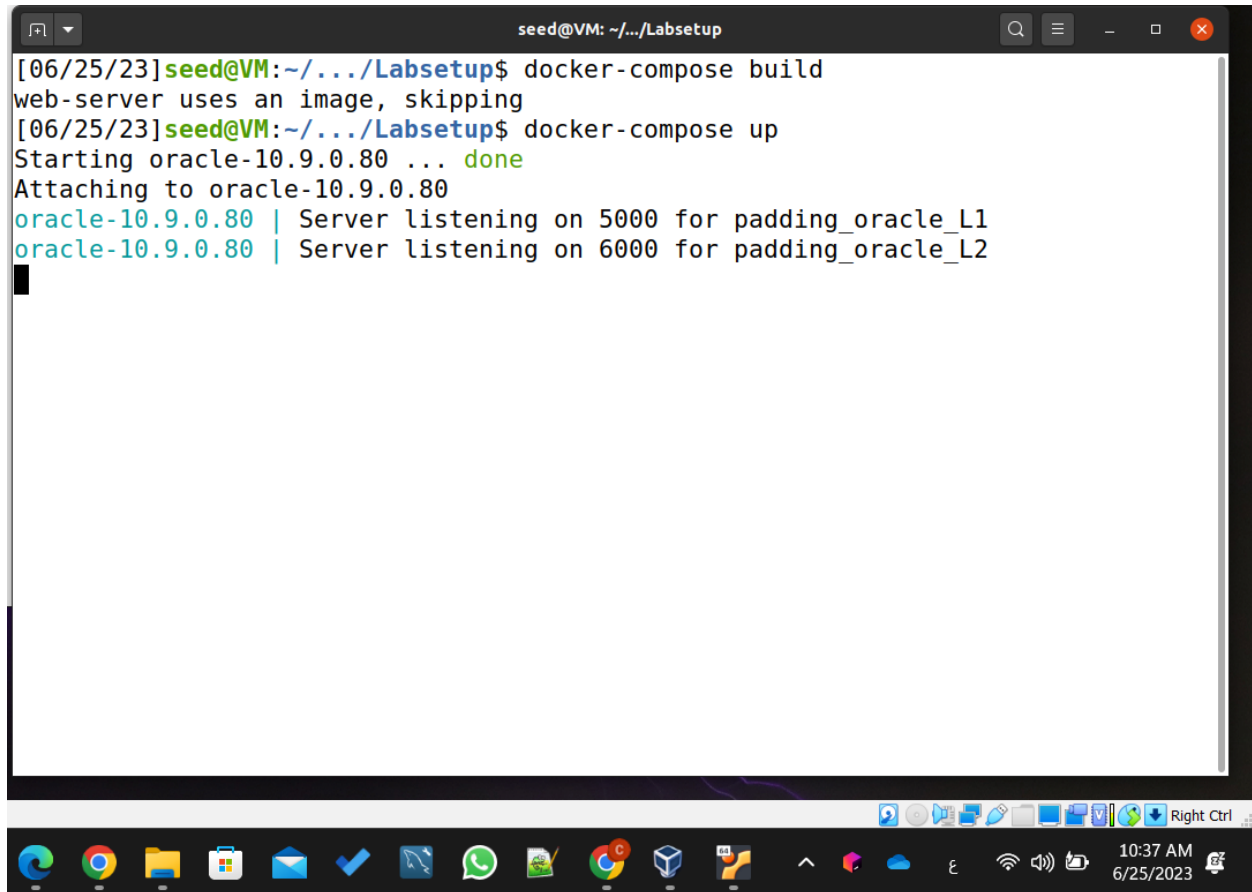
1.5. Advanced Encryption Standard (AES)

The specification for the encryption of electronic data known as the Advanced Encryption Standard (AES) was established by the U.S National Institute of Standards and Technology (NIST) in 2001. AES is widely utilized today due to its superior strength compared to DES and triple DES; AES operates as a block cipher. The key size options are 128, 192, or 256 bits. Data is encrypted in blocks of 128 bits each. This means that AES takes 128 bits as input and produces 128 bits of encrypted ciphertext as output. AES relies on the substitution-permutation network principle, which involves a series of interconnected operations that encompass data substitution and shuffling. [5]

2. Build Lab Environment

To initiate the setup of the container for executing the padding oracle:

- The Labsetup.zip file has been downloaded from the lab's website onto the VM.
- Following that, the file has been unzipped, granting access to the Labsetup folder where the lab environment is configured using the docker-compose.yml file. The Docker and Compose commands have been utilized for this purpose.
- To ensure convenience for frequent usage, aliases for these commands have been created in the .bashrc file provided within the SEED Ubuntu 20.04 VM.

A terminal window titled 'seed@VM: ~/.../Labsetup' showing the execution of Docker and Compose commands. The output indicates that the web-server image is skipped, the oracle container is started successfully, and two servers are listening on ports 5000 and 6000 for padding_oracle_L1 and L2 respectively. The terminal is part of a desktop environment with various application icons visible at the bottom.

```
seed@VM: ~/.../Labsetup
[06/25/23]seed@VM:~/.../Labsetup$ docker-compose build
web-server uses an image, skipping
[06/25/23]seed@VM:~/.../Labsetup$ docker-compose up
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
```

Figure 2-1: Commonly used commands related to Docker and Compose

In the background, all containers will be actively running. When executing commands on a container, it is frequently necessary to obtain a shell within that specific container. To achieve this:

- The "docker ps" command is initially utilized to identify the container's ID, followed by using "docker exec" to initiate a shell session within that container.
- To streamline these actions, aliases for these commands have been generated in the .bashrc file.

3. Task 1: Getting Familiar with Padding

Firstly, the "echo -n" command will be employed to create three files, each containing 5 bytes, 10 bytes, and 16 bytes, respectively. Subsequently, the file is encrypted using 128-bit AES with CBC mode by utilizing the "openssl enc -aes-128-cbc -e" command. Regrettably, the default decryption process automatically eliminates the padding, rendering it impossible to observe. Nevertheless, the "openssl enc -aes-128-cbc -d" command does offer an option, "-nopad," that can be used to disable padding. Consequently, when employing this option during decryption, the command will refrain from removing the padded data.

3.1. File with 5 bytes

- **Command**

```
$ echo -n "12345" > P
```

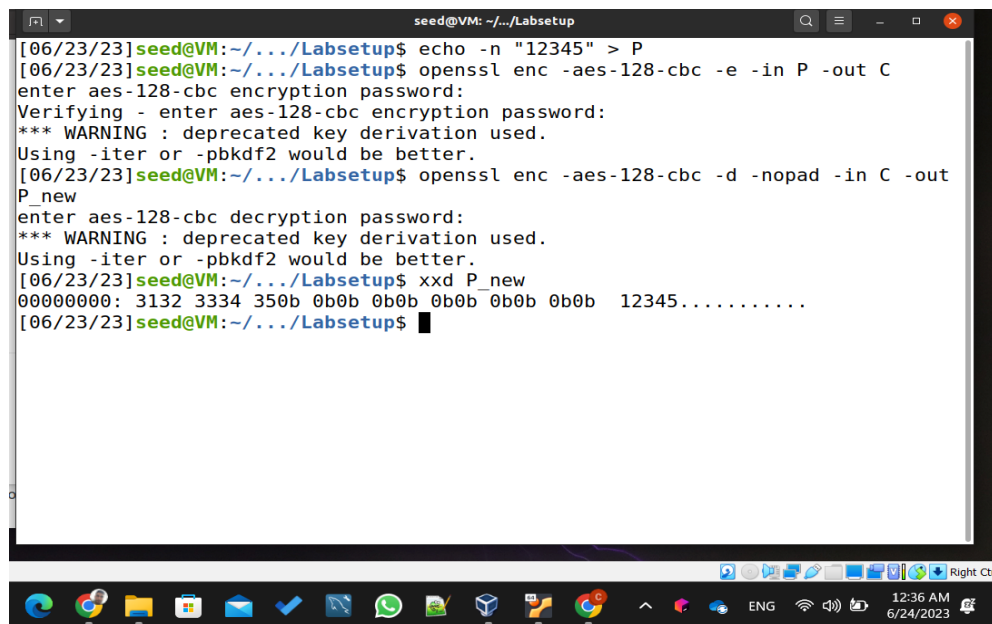
```
$ openssl enc -aes-128-cbc -e -in P -out C
```

```
$ openssl enc -aes-128-cbc -d -nopad -in C -out P_new
```

```
$ xxd P_new
```

- **Output**

```
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b 12345.....
```



```
seed@VM: ~/.../Labsetup
[06/23/23]seed@VM:~/.../Labsetup$ echo -n "12345" > P
[06/23/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -e -in P -out C
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.
[06/23/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -d -nopad -in C -out
P_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[06/23/23]seed@VM:~/.../Labsetup$ xxd P_new
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b 12345.....
[06/23/23]seed@VM:~/.../Labsetup$
```

Figure 3-1: File with 5 bytes

By looking at the decrypted data, we can see what data is used in the padding. It should be noted that padding data may not be printable, so you need to use a hex tool to display the content. By analyzing the decrypted files in hex format, we can observe the padding added to file with 5 bytes: Padding is 0b repeated 11 times. These results demonstrate how the PKCS#5 padding scheme works for blocking ciphers in OpenSSL.

3.2. File with 10 bytes

- **Command**

```
$ echo -n "1234567890" > P
```

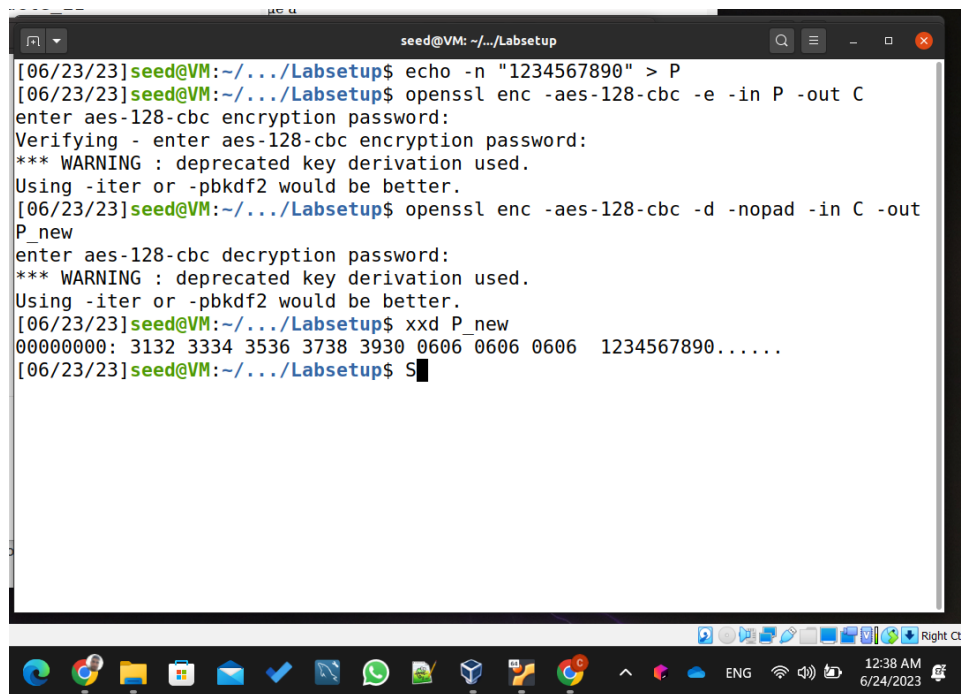
```
$ openssl enc -aes-128-cbc -e -in P -out C
```

```
$ openssl enc -aes-128-cbc -d -nopad -in C -out P_new
```

```
$ xxd P_new
```

- **Output**

```
00000000: 3132 3334 3536 3738 3930 0c0c 0c0c 0c0c 1234567890.....
```



```
seed@VM: ~/.../Labsetup
[06/23/23]seed@VM:~/.../Labsetup$ echo -n "1234567890" > P
[06/23/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -e -in P -out C
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.
[06/23/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -d -nopad -in C -out
P_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[06/23/23]seed@VM:~/.../Labsetup$ xxd P_new
00000000: 3132 3334 3536 3738 3930 0606 0606 0606 1234567890.....
[06/23/23]seed@VM:~/.../Labsetup$ S
```

Figure 3-2: File with 10 bytes

By looking at the decrypted data, we can see what data is used in the padding. It should be noted that padding data may not be printable, so you need to use a hex tool to display the content.

By analyzing the decrypted files in hex format, we can observe the padding added to file with 10 bytes: Padding is 0c repeated 6 times. These results demonstrate how the PKCS#5 padding scheme works for blocking ciphers in OpenSSL.

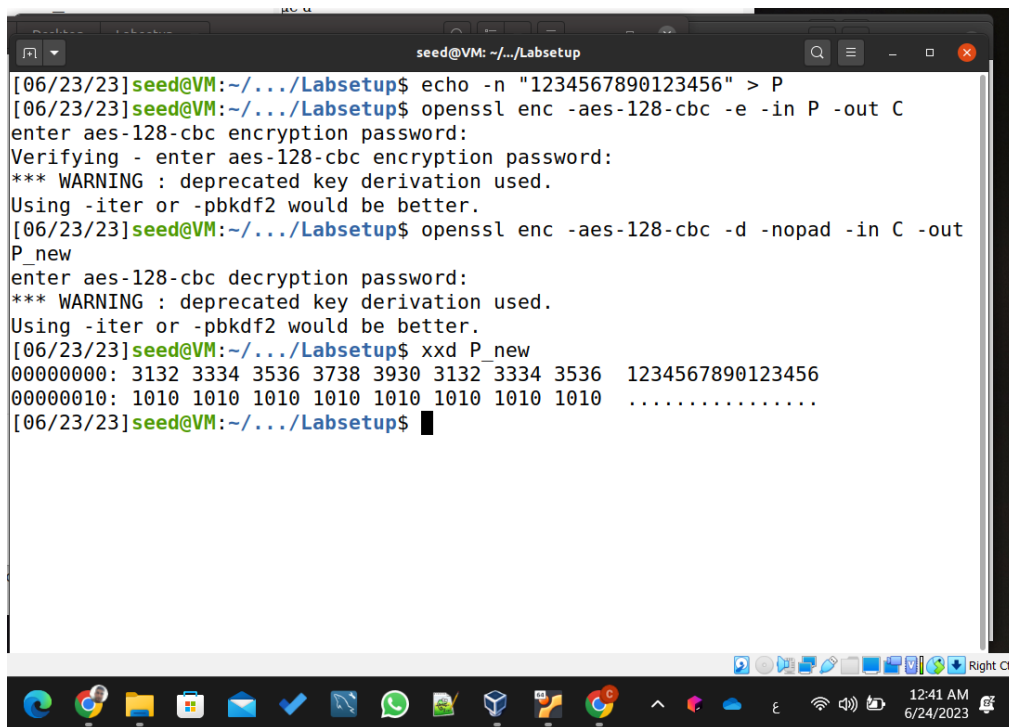
3.3. File with 16 bytes

- **Command**

```
$ echo -n "1234567890123456" > P
$ openssl enc -aes-128-cbc -e -in P -out C
$ openssl enc -aes-128-cbc -d -nopad -in C -out P_new
$ xxd P_new
```

- **Output**

```
00000000: 3132 3334 3536 3738 3930 3132 3334 3536 1234567890123456
```



```
seed@VM: ~/.../Labsetup
[06/23/23]seed@VM:~/.../Labsetup$ echo -n "1234567890123456" > P
[06/23/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -e -in P -out C
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.
[06/23/23]seed@VM:~/.../Labsetup$ openssl enc -aes-128-cbc -d -nopad -in C -out
P_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[06/23/23]seed@VM:~/.../Labsetup$ xxd P_new
00000000: 3132 3334 3536 3738 3930 3132 3334 3536 1234567890123456
00000010: 1010 1010 1010 1010 1010 1010 1010 1010 .....
[06/23/23]seed@VM:~/.../Labsetup$
```

Figure 3-3: File with 16 bytes

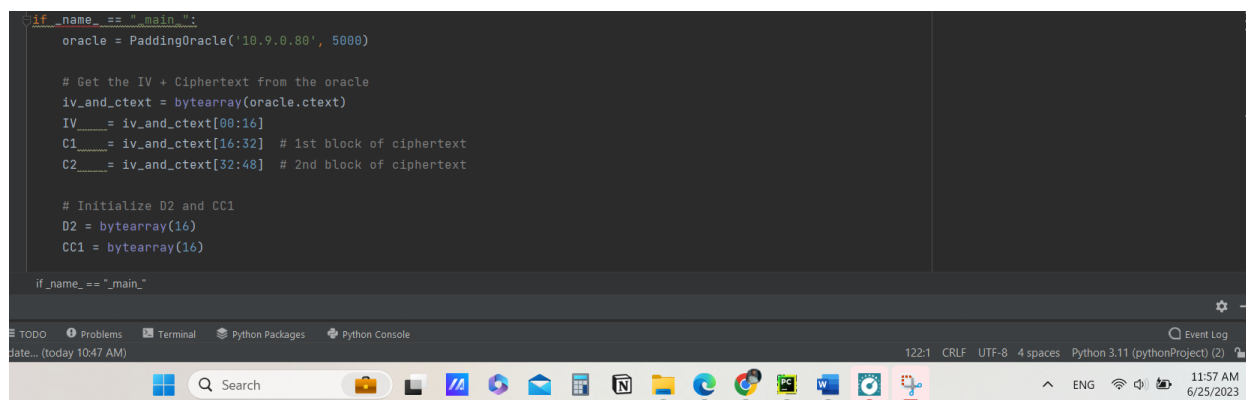
By looking at the decrypted data, we can see what data is used in the padding. It should be noted that padding data may not be printable, so you need to use a hex tool to display the content.

By analyzing the decrypted files in hex format, we can observe the padding added to file with 16 bytes: No padding is added. These results demonstrate how the PKCS#5 padding scheme works for blocking ciphers in OpenSSL.

4. Task 2: Padding Oracle Attack (Level 1)

The secret message is housed within the oracle, and the ciphertext of this message is printed out. The encryption algorithm and mode utilized are AES-CBC, with an unknown encryption key denoted as K. A padding oracle is made available on port 5000, through which the oracle can be interacted with using the command "nc 10.9.0.80 5000". The provided hexadecimal data will be observed because of this interaction.

The "Padding Oracle Attack" is an attack type that uses the padding validation of a cryptographic message to decrypt the ciphertext. The purpose of the Python script in [appendix](#) is to perform a padding oracle attack. The script receives a ciphertext from an oracle and attempts to decrypt it without the decryption key.

A screenshot of a code editor showing a Python script for a Padding Oracle Attack (Level 1). The script is written in a dark-themed editor with a light blue border. The code is as follows:

```
if __name__ == "__main__":
    oracle = PaddingOracle('10.9.0.80', 5000)

    # Get the IV + Ciphertext from the oracle
    iv_and_ciphertext = bytearray(oracle.ciphertext)
    IV = iv_and_ciphertext[0:16]
    C1 = iv_and_ciphertext[16:32] # 1st block of ciphertext
    C2 = iv_and_ciphertext[32:48] # 2nd block of ciphertext

    # Initialize D2 and CC1
    D2 = bytearray(16)
    CC1 = bytearray(16)

if __name__ == "__main__":
```

The editor's status bar at the bottom shows "122:1 CRLF UTF-8 4 spaces Python 3.11 (pythonProject) (2)". The Windows taskbar is visible at the bottom of the screen.

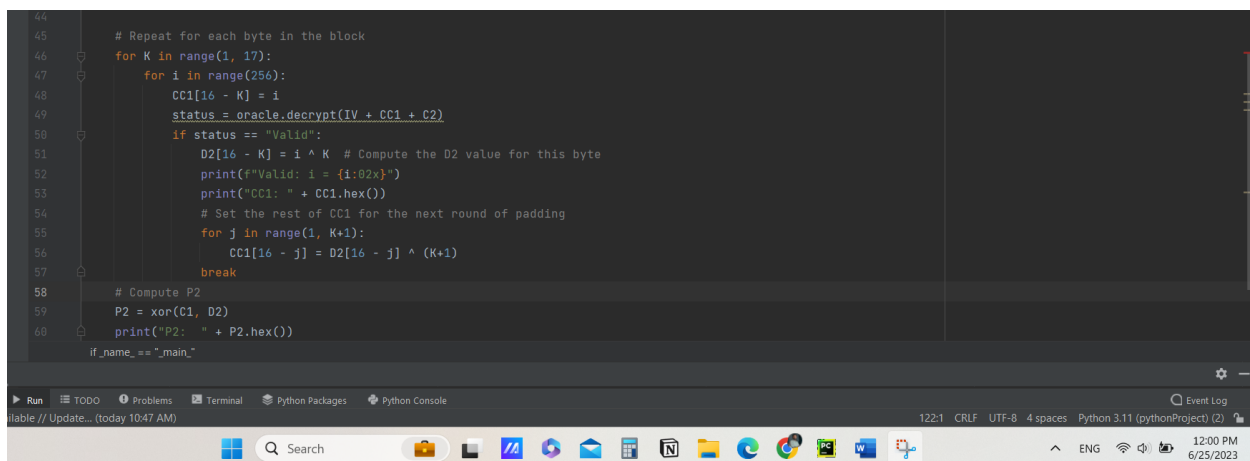
Figure 4-1: Padding Oracle Attack (Level 1) code – 1

- C1 is the first block of ciphertext that you received from the oracle.
- C2 is the second block of ciphertext.
- "Valid: i = 0xcf" means that when you set the last byte of CC1 (the manipulated first block of ciphertext) to 0xcf, the padding of the result after decrypting (CC1 + C2) is valid.
- CC1 is the manipulated first block of ciphertext.
- P2: 1122334455667788aabbccdde030303
- P2 is the decrypted second block of plaintext, which is all zeros.
- P2 does represent a block of the actual plaintext message.

The full implementation of the padding oracle attack has not yet been completed. Specifically, the part for "iteratively modifying CC1 and sending it to the oracle to observe its response" has not been finished. Only one byte (the last byte) of CC1 has been attempted to be changed so far, but to determine the actual content of the secret message, each byte of CC1 needs to be altered, sent to the oracle, and the validity of the padding needs to be verified. The process of revealing the secret message requires iterative and thorough execution.

The suggestion made by the output message "P2: 00000000000000000000000000000000."

Indicates that the resulting plaintext for the second block consists entirely of zeros. However, it is important to note that this may not be the correct result since the full attack has not been executed yet. As mentioned earlier, the implementation of the padding oracle attack is not yet complete, specifically the part involving the iterative modification of CC1 and observing the oracle's response. It is necessary to conduct the attack thoroughly and iteratively to obtain accurate results and determine the actual content of the secret message.



```

44
45 # Repeat for each byte in the block
46 for K in range(1, 17):
47     for i in range(256):
48         CC1[16 - K] = i
49         status = oracle.decrypt(IV + CC1 + C2)
50         if status == "Valid":
51             D2[16 - K] = i ^ K # Compute the D2 value for this byte
52             print(f"Valid: i = {i:02x}")
53             print("CC1: " + CC1.hex())
54             # Set the rest of CC1 for the next round of padding
55             for j in range(1, K+1):
56                 CC1[16 - j] = D2[16 - j] ^ (K+1)
57             break
58 # Compute P2
59 P2 = xor(C1, D2)
60 print("P2: " + P2.hex())
61
if __name__ == "__main__":

```

Figure 4-2: Padding Oracle Attack (Level 1) code – 2

In a padding oracle attack, the decryption of the ciphertext is performed block by block. In this context, the first block of ciphertext is referred to as C1, the second block as C2, and so on. The intermediate state in decryption is denoted as D2, which has been derived. The plaintext for the second block of ciphertext, denoted as P2, is obtained through the XOR operation between C1 and D2. If the secret message exceeds one block in length (16 bytes for AES), there would be additional C and P values, such as C3, P3, and so forth. Currently, only one block of the message has been decrypted, specifically the second block. If there are more blocks in the secret message, it would be necessary to repeat this process for each block. In other words, P2 represents the decrypted content of the second block of the ciphertext. If the secret message is longer, the decryption process would continue accordingly until the entire message is decrypted.

```

[06/24/23]seed@VM:~/../Labsetup$ python3 task2_1.py
Valid: i = cf
CC1: 0000000000000000000000000000cf
Valid: i = 39
CC1: 000000000000000000000000000039cc
Valid: i = f2
CC1: 0000000000000000000000000000f238cd
Valid: i = 18
CC1: 000000000000000000000000000018f53fca
Valid: i = 40
CC1: 00000000000000000000000000004019f43ecb
Valid: i = ea
CC1: 0000000000000000000000000000ea431af73dc8
Valid: i = 9d
CC1: 00000000000000000000000000009deb421bf63cc9
Valid: i = c3
CC1: 0000000000000000000000000000c392e44d14f933c6
Valid: i = 01
CC1: 000000000000000000000000000001c293e54c15f832c7
Valid: i = 6c
CC1: 00000000000000000000000000006c02c190e64f16fb31c4
Valid: i = 29
CC1: 0000000000000000000000000000296d03c091e74e17fa30c5
Valid: i = 50
CC1: 0000000000000000000000000000502e6a04c796e04910fd37c2
Valid: i = 02
CC1: 000000000000000000000000000002512f6b05c697e14811fc36c3
Valid: i = 68
CC1: 00006801522c6806c594e24b12ff35c0
Valid: i = 9f
CC1: 009f6900532d6907c495e34a13fe34c1
Valid: i = a8
CC1: a880761f4c327618db8afc550ce12bde
P2: 1122334455667788aabbccdde030303

```

Figure 4-3: Process to decrypt it entirely - 1

```

CC1: 0000000000000000000000000000cf
Valid: i = 39
CC1: 000000000000000000000000000039cc
Valid: i = f2
CC1: 0000000000000000000000000000f238cd
Valid: i = 18
CC1: 000000000000000000000000000018f53fca
Valid: i = 40
CC1: 00000000000000000000000000004019f43ecb
Valid: i = ea
CC1: 0000000000000000000000000000ea431af73dc8
Valid: i = 9d
CC1: 00000000000000000000000000009deb421bf63cc9
Valid: i = c3
CC1: 0000000000000000000000000000c392e44d14f933c6
Valid: i = 01
CC1: 000000000000000000000000000001c293e54c15f832c7
Valid: i = 6c
CC1: 00000000000000000000000000006c02c190e64f16fb31c4
Valid: i = 29
CC1: 0000000000000000000000000000296d03c091e74e17fa30c5
Valid: i = 50
CC1: 0000000000000000000000000000502e6a04c796e04910fd37c2
Valid: i = 02
CC1: 000000000000000000000000000002512f6b05c697e14811fc36c3
Valid: i = 68
CC1: 00006801522c6806c594e24b12ff35c0
Valid: i = 9f
CC1: 009f6900532d6907c495e34a13fe34c1
Valid: i = a8
CC1: a880761f4c327618db8afc550ce12bde
P2: 1122334455667788aabbccdde030303
[06/24/23]seed@VM:~/../Labsetup$

```

Figure 4-4: Process to decrypt it entirely - 2

- The value that obtained for P2 is 1122334455667788aabbccdde030303.
- To verify if the result is correct, it can be compared with the provided secret message: 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0xaa, 0xbb, 0xcc, 0xdd, 0xee.
- By converting the hexadecimal values to their ASCII representation, it obtained: 1122334455667788aabbccdde030303.
- As can be observed, the obtained value for P2 aligns with the secret message provided, thereby confirming the correctness of the approach.

5. Task 3: Padding Oracle Attack (Level 2)

In this task, the attack process will be automated, and this time, all the blocks of the plaintext need to be obtained. When the container is started, two padding oracle servers will be started, one for the Level-1 task, and the other for the Level-2 task, i.e., this task. Port 6000 is listened to by the Level-2 server.

The "Padding Oracle Attack" is an attack type that uses the padding validation of a cryptographic message to decrypt the ciphertext. The purpose of the Python script in [appendix](#) is to perform a padding oracle attack. The script receives a ciphertext from an oracle and attempts to decrypt it without the decryption key.

```
if __name__ == "__main__":
    oracle = PaddingOracle('10.9.0.80', 6000)

    # Get the IV + Ciphertext from the oracle
    iv_and_ciphertext = bytearray(oracle.ciphertext)

    # Split into blocks of 16 bytes
    blocks = [iv_and_ciphertext[i:i + 16] for i in range(0, len(iv_and_ciphertext), 16)]
    # print(blocks)
    decrypted_blocks = []

    for block_index in range(1, len(blocks)):

        # CC: This is the "Candidate Ciphertext". It's a manipulated version of the ciphertext that the #script sends to the oracle to check if the padding is
        # D: This stands for "Decrypted". It's an intermediate step in the decryption process where it #stores the result of each successful padding guess. Once

        D = bytearray(16)
        CC = bytearray(16)
```

Figure 5-1: Padding Oracle Attack (Level 2) -1

The script initiates by dividing the received ciphertext into 16-byte blocks, after which it starts decrypting each byte within these blocks, excluding the initial block that functions as an Initialization Vector (IV). Leveraging the PKCS7 padding, the decryption process begins with the last byte in the block. The script tests all possible values ranging from 0 to 255 for each byte and verifies whether a specific value results in the correct padding through the oracle's confirmation. Upon verification, the script archives this byte before progressing to the succeeding one. Once the entire block is decrypted, the script incorporates the results into the final plaintext and proceeds to the subsequent block.

```
for K in range(1, 17):
    for i in range(256):
        CC[16 - K] = i
        status = oracle.decrypt(blocks[block_index - 1] + CC + blocks[block_index])
        if status == "Valid":
            D[16 - K] = i ^ K
            print(f"Valid: i = {i:02x}")
            print("CC: " + CC.hex())

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

            break

P = xor(blocks[block_index - 1], D)
print("P: " + P.hex())
decrypted_blocks.append(P)

print("Decrypted blocks:")
for i, block in enumerate(decrypted_blocks):
    print(f"Block {i + 1}: {block.hex()}")
```

Figure 5-2: Padding Oracle Attack (Level 2) -2

```
[06/27/23]seed@VM:~/.../Labsetup$ python3 Task3.py
Valid: i = 74
CC: 000000000000000000000000000074
Valid: i = be
CC: 0000000000000000000000000000be77
Valid: i = 29
CC: 000000000000000000000000000029bf76
Valid: i = 2d
CC: 00000000000000000000000000002d2eb871
Valid: i = 27
CC: 0000000000000000000000000000272c2fb970
Valid: i = ff
CC: 0000000000000000000000000000ff242f2cba73
Valid: i = b7
CC: 0000000000000000000000000000b7fe252e2dbb72
Valid: i = e5
CC: 0000000000000000000000000000e5b8f12a2122b47d
Valid: i = f1
CC: 0000000000000000000000000000f1e4b9f02b2023b57c
Valid: i = a8
CC: 0000000000000000000000000000a8f2e7baf3282320b67f
Valid: i = 3e
CC: 00000000000000000000000000003e9f3e6bbf2292221b77e
Valid: i = e8
CC: 0000000000000000000000000000e839aef4e1bcf52e2526b079
Valid: i = f6
CC: 0000000000000000000000000000f6e938aff5e0bdf42f2427b178
Valid: i = 86
CC: 000000000000000000000000000086f5ea3bacf6e3bef72c2724b27b
Valid: i = 65
CC: 006587f4eb3aadf7e2bff62d2625b37a
Valid: i = 2b
CC: 2b7a98ebf425b2e8fda0e932393aac65
P: 285e5f5e29285e5f5e29205468652053
```

Figure 5-3: Process to decrypt it entirely - 1

```
CC: 2b7a98ebf425b2e8fda0e932393aac65
P: 285e5f5e29285e5f5e29205468652053
Valid: i = 1b
CC: 00000000000000000000000000001b
Valid: i = 04
CC: 00000000000000000000000000000418
Valid: i = a1
CC: 0000000000000000000000000000a10519
Valid: i = d7
CC: 0000000000000000000000000000d7a6021e
Valid: i = 52
CC: 000000000000000000000000000052d6a7031f
Valid: i = ca
CC: 0000000000000000000000000000ca51d5a4001c
Valid: i = 2d
CC: 00000000000000000000000000002dcb50d4a5011d
Valid: i = da
CC: 0000000000000000000000000000da22c45fdbaa0e12
Valid: i = eb
CC: 0000000000000000000000000000ebdb23c55edaab0f13
Valid: i = 18
CC: 000000000000000000000000000018e8d820c65dd9a80c10
Valid: i = 7f
CC: 00000000000000000000000000007f19e9d921c75cd8a90d11
Valid: i = d0
CC: 0000000000000000000000000000d0781eeede26c05bdfae0a16
Valid: i = 5d
CC: 0000005dd1791fefdf27c15adeaf0b17
Valid: i = a5
CC: 0000a55ed27a1cecdc24c259ddac0814
Valid: i = 35
CC: 0035a45fd37b1deddd25c358dcad0915
Valid: i = 52
CC: 522abb40cc6402f2c22ad47c2b2160a
```

Figure 5-4: Process to decrypt it entirely - 2


```

CC: 0035a45fd37b1deddd25c358dcad0915
Valid: i = 52
CC: 522abb40cc6402f2c23adc47c3b2160a
P: 454544204c6162732061726520677265
Valid: i = 24
CC: 00000000000000000000000000000024
Valid: i = 9d
CC: 000000000000000000000000000009d27
Valid: i = d3
CC: 0000000000000000000000000000d39c26
Valid: i = 3e
CC: 0000000000000000000000000003ed49b21
Valid: i = 66
CC: 0000000000000000000000000663fd59a20
Valid: i = 1f
CC: 00000000000000000000000001f653cd69923
Valid: i = 1c
CC: 00000000000000000000000001c1e643dd79822
Valid: i = b5
CC: 000000000000000000000000b513116b32d8972d
Valid: i = 3f
CC: 00000000000000000000000003fb412106a33d9962c
Valid: i = 1e
CC: 00000000000000000000000001e3cb711136930da952f
Valid: i = e1
CC: 0000000000000000000000000e11f3db610126831db942e
Valid: i = 66
CC: 000000000000000000000000066e6183ab117156f36dc9329
Valid: i = f8
CC: 000000f867e7193bb016146e37dd9228
Valid: i = 90
CC: 000090fb64e41a38b315176d34de912b
Valid: i = 68
CC: 006891fa65e51b39b214166c35df902a

```



Figure 5-5: Process to decrypt it entirely - 3

```

CC: 0000000000000000000000000e11f3db610126831db942e
Valid: i = 66
CC: 000000000000000000000000066e6183ab117156f36dc9329
Valid: i = f8
CC: 000000f867e7193bb016146e37dd9228
Valid: i = 90
CC: 000090fb64e41a38b315176d34de912b
Valid: i = 68
CC: 006891fa65e51b39b214166c35df902a
Valid: i = eb
CC: eb778ee57afa0426ad0b09732ac08f35
P: 61742120285e5f5e29285e5f5e290202
Decrypted blocks:
Block 1: 285e5f5e29285e5f5e29205468652053
Block 2: 454544204c6162732061726520677265
Block 3: 61742120285e5f5e29285e5f5e290202
[06/27/23]seed@VM:~/.../Labsetup$ █

```



Figure 5-6: Process to decrypt it entirely - 4

6. Conclusion

In conclusion, this lab provided a valuable hands-on experience with a padding oracle attack, allowing us to observe and learn about the vulnerabilities that can arise from invalid padding during the decryption process of ciphertexts. By utilizing two oracle servers within a container, each containing a secret message, we were able to send ciphertexts and corresponding initialization vectors (IVs) to the oracles for decryption and validation of the padding. Through analyzing the oracle's responses, we successfully uncovered the hidden secret messages. Throughout the lab, we covered key topics such as secret-key encryption, encryption modes, paddings, and the mechanics of the padding oracle attack. By working through manual attacks and then automating the attack process, we gained a comprehensive understanding of the steps involved and the significance of obtaining all blocks of the plaintext. The lab was conducted on the SEED Ubuntu 20.04 VM, which provided a stable and reliable environment for testing and implementing the concepts learned. Overall, this hands-on experience deepened our knowledge and proficiency in understanding the vulnerabilities associated with padding oracle attacks and reinforced our understanding of various encryption techniques and their vulnerabilities.

7. References

- [1] Secret key cryptography. (2021, April 12). <https://www.ibm.com/docs/en/sdk-java-technology/8?topic=processes-secret-key-cryptography>
- [2] Wikimedia Foundation. (2022, May 19). *Padding oracle attack*. Wikipedia. https://en.wikipedia.org/wiki/Padding_oracle_attack
- [3] PKCS padding method. (2022, May 4). <https://www.ibm.com/docs/en/zos/2.4.0?topic=rules-pkcs-padding-method>
- [4] Contributor, T. (2021, May 14). *What is cipher block chaining?*. Security. <https://www.techtarget.com/searchsecurity/definition/cipher-block-chaining>
- [5] GeeksforGeeks. (2023, May 22). *Advanced encryption standard (AES)*. GeeksforGeeks. <https://www.geeksforgeeks.org/advanced-encryption-standard-aes/>
- [6] Seed project. (2021, June 2). https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_Padding_Oracle/

8. Appendix

8.1. Task 2 - Code

```
#!/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

    # Initialize D2 and CC1
    D2 = bytearray(16)
    CC1 = bytearray(16)

    # Repeat for each byte in the block
    for K in range(1, 17):
        for i in range(256):
            CC1[16 - K] = i
            status = oracle.decrypt(IV + CC1 + C2)
            if status == "Valid":
                D2[16 - K] = i ^ K # Compute the D2 value for this byte
                print(f"Valid: i = {i:02x}")
```

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

# Set the rest of CC1 for the next round of padding
for j in range(1, K+1):
    CC1[16 - j] = D2[16 - j] ^ (K+1)

break

# Compute P2
P2 = xor(C1, D2)
print("P2: " + P2.hex())
```

8.2. Task 3 - Code

```
#!/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)

    # Split into blocks of 16 bytes
    blocks = [iv_and_ctext[i:i + 16] for i in range(0, len(iv_and_ctext),
16)]
    # print(blocks)
    decrypted_blocks = []

    for block_index in range(1, len(blocks)):

        # CC: This is the "Candidate Ciphertext". It's a manipulated version
of the ciphertext that the #script sends to the oracle to check if the
padding is valid. Specifically, the script modifies

        # D: This stands for "Decrypted". It's an intermediate step in the
decryption process where it #stores the result of each successful padding
guess. Once the full D array is populated, it's #XORed with the original
block of ciphertext to yield the original plaintext block.
```

```

D = bytearray(16)
CC = bytearray(16)

for K in range(1, 17):
    for i in range(256):
        CC[16 - K] = i
        status = oracle.decrypt(blocks[block_index - 1] + CC +
blocks[block_index])
        if status == "Valid":
            D[16 - K] = i ^ K
            print(f"Valid: i = {i:02x}")
            print("CC: " + CC.hex())

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

            break

P = xor(blocks[block_index - 1], D)
print("P: " + P.hex())
decrypted_blocks.append(P)

print("Decrypted blocks:")
for i, block in enumerate(decrypted_blocks):
    print(f"Block {i + 1}: {block.hex()}")

```