



Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Cryptography lab

Padding Oracle Attack Lab

Prepared by:

Jana Herzallah 1201139

Lana Thabit 1200071

Instructor: Dr.Ahmad Alsaadeh

Section: 1

Date: 24/6/2023

Abstract

The Padding Oracle Attack Lab is designed to provide students with a hands-on experience of a cryptographic attack known as the padding oracle attack. This attack exploits the behavior of systems that verify the validity of padding when decrypting ciphertext, hence making analysis over the whole system. The lab involves two oracle servers running in a container which were pulled in an image in the docker environment, each containing a secret message. We as students are given the ciphertext and the initialization vector (IV) but not the plaintext or encryption key. So, the whole process was mainly around analyzing the oracle's response of the server after sending a cipher text regarding the validity of padding so we can discover the message hidden in the oracle. The lab environment is set up using a container, and instructions are provided for building and running the environment using Docker and Compose commands. The lab tasks involve understanding padding schemes, manually deriving plaintext using the padding oracle attack, and using iterative processes to determine the values of decrypted blocks.

Contents

Abstract.....	2
Table of figures:	4
Introduction:	5
1. Setting the lab environment	6
2. Task 1	8
a) File 5 bytes	8
b) file 10 bytes.....	9
c) file 16 bytes.....	10
3. Task 2 Padding Oracle Attack (Level 1)	11
part 1: The Oracle Setup	11
Part 2- guessing the message:	14
Explanation:	22
4. Task 3: Padding Oracle Attack (Level 2)	24
First part.....	24
Second part:	26
Explanation:	28
Conclusion	33
We found it really interesting	34
Appendix:	35
task1.....	35
Taskk2.py	37
Task3-automatedtask2.py	40
Task3.py	42

Table of figures:

Figure 1:powershell updating the wsl.....	6
Figure 2 : creating a new dev environment and pulling the required image	7
Figure 3 : creating a 5 bytes file through the terminal	8
Figure 4: : creating a 10 bytes file through the terminal	9
Figure 5 : : creating a 16 bytes file through the terminal	10
Figure 6: setting the docker compose file for task2	14
Figure 7: installing puthon3 on the device through the docker terminal.....	17
Figure 8 : running the manual_attck file after installing python3	18
Figure 9 : zoomed in picture of the output.....	18
Figure 10 : running Task2.py file	19
Figure 11 : screenshot of Task2.py on my device	20
Figure 12 : first 3 blocks from k=1 to k=3.....	21
Figure 13 :second 3 blocks from k=4 to k=6	21
Figure 14 : desired blocks 6 blocks of D2 manually runned	22
Figure 15 : running Task3-automatedtask2 through the docker terminal	24
Figure 16 : 6 bytes of D2 automatically runned.....	25
Figure 17:Automated task2	26
Figure 18 : connecting to port 6000 through the docker terminal.....	26
Figure 19 : Task3.py file running in visual studio	27

Introduction:

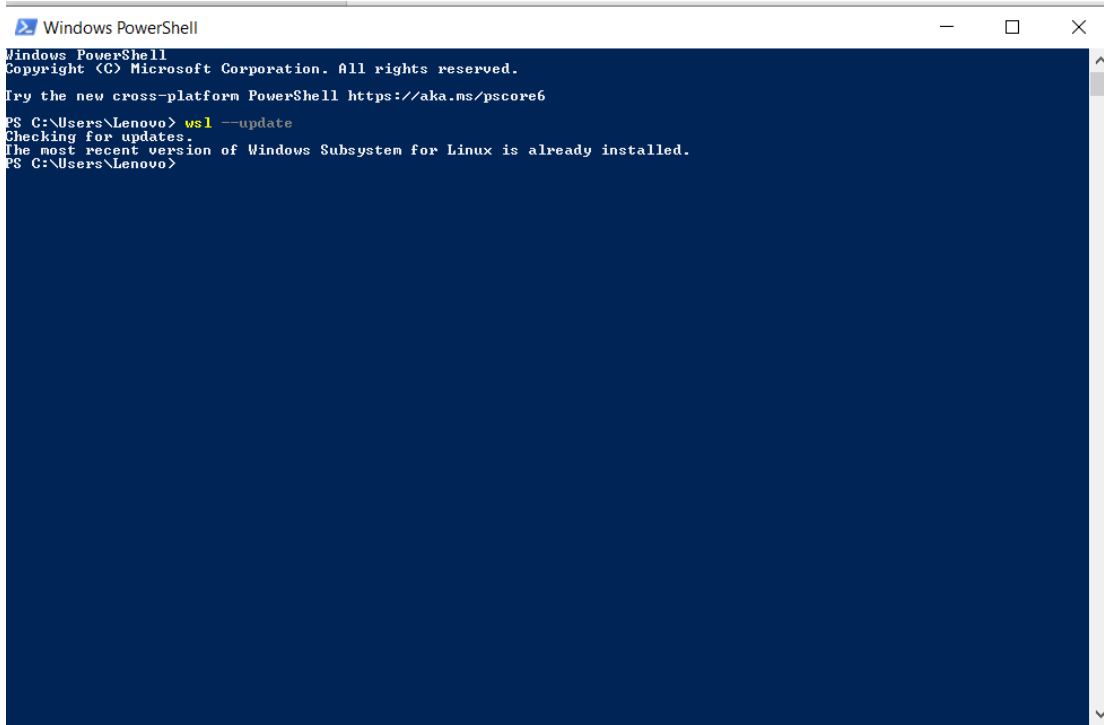
In today's world of cryptography, we can find systems that are designed to protect sensitive information that may have harmful effects if they get discovered. However, some of these systems have vulnerabilities that attackers can find their ways into. One such vulnerability is the padding oracle attack, which takes advantage of systems that check the validity of padding during decryption. This lab aims to teach us students about secret-key encryption, encryption modes and paddings, as well as the steps involved in executing a successful padding oracle attack with hands-on experience of the padding oracle attack and its implications.

This lab was designed to work on two oracle servers that contains two secrets. We will be able to interact with the servers using encrypted message and the initialization vector (IV). And our task was to decrypt the secret messages by utilizing information obtained from the padding oracle.

The lab consists of three tasks that progressively build to achieve the objectives mentioned above, In the first task, we will create files of different lengths and analyze the padding applied to each file. Moving on to the second task, we will focus on decrypting the secret message by gradually determining parts of an intermediate array called D2. In the final task, we will automate the attack process and aim to retrieve all the blocks of the secret message.

1. Setting the lab environment

We have downloaded the docker desktop file and then installed it, then we had to update the windows subsystem for Linux (WSL) which is a feature of the Windows operating system that enables you to run a Linux file system, along with Linux command-line tools and GUI apps, directly on Windows, alongside your traditional Windows desktop and apps correctly through this command “wsl—update “on PowerShell.

A screenshot of a Windows PowerShell window. The title bar reads "Windows PowerShell". The terminal text is as follows:

```
Windows PowerShell  
Copyright (C) Microsoft Corporation. All rights reserved.  
Try the new cross-platform PowerShell https://aka.ms/pscore6  
PS C:\Users\Lenovo> wsl --update  
Checking for updates.  
The most recent version of Windows Subsystem for Linux is already installed.  
PS C:\Users\Lenovo>
```

Figure 1:powershell updating the wsl

Then we created a new dev environment and gave it the directory of the labsetup file given in order to pull the image that has 2 containers which are the [oracle-10.9.0.80](#) and the app container that we used to type in the commands using the internal terminal.

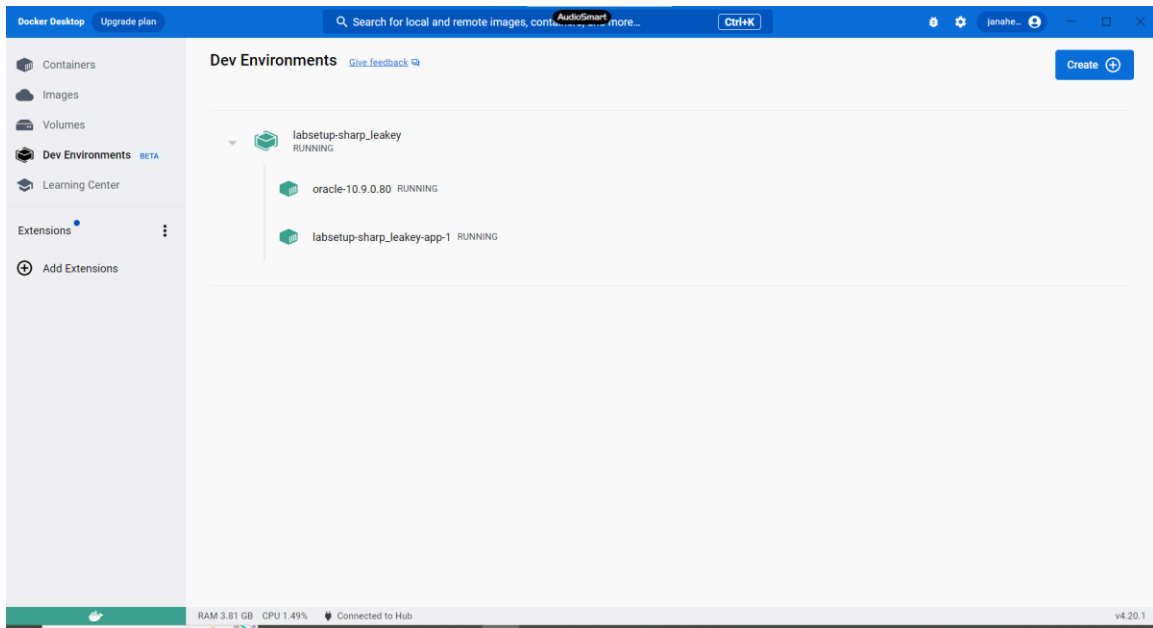
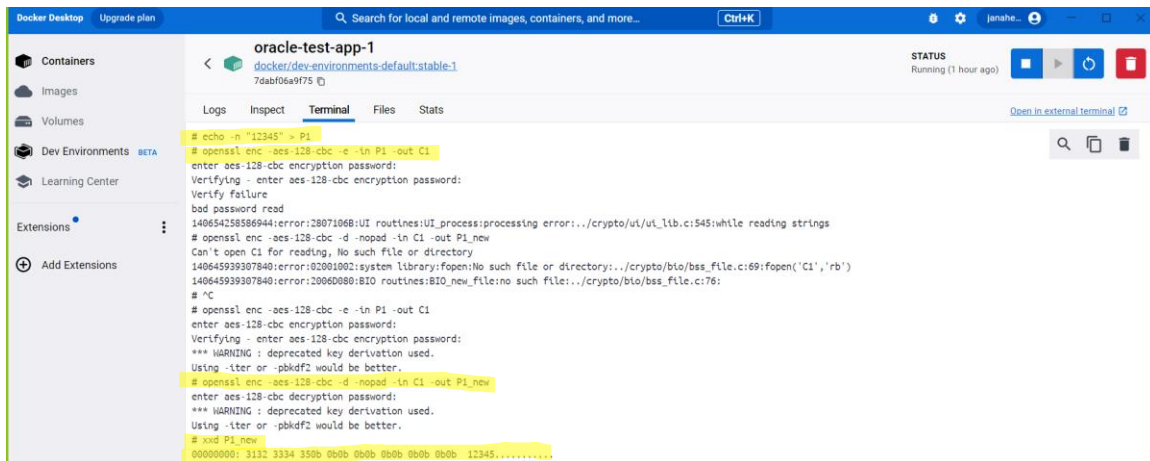


Figure 2 : creating a new dev environment and pulling the required image

2. Task 1

a) File 5 bytes



```
oracle-test-app-1
docker/dev-environments/default:stable-1
7dabf06a9175

# echo -n "12345" > P1
# openssl enc -aes-128-cbc -e -in P1 -out C1
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
Verify failure
bad password read
140654258586944:error:2807106B:UI routines:UI_process:processing error:../crypto/ui/ui_lib.c:545:while reading strings
# openssl enc -aes-128-cbc -d -nopad -in C1 -out P1_new
Can't open C1 for reading, No such file or directory
140645939307840:error:82001002:system library:fopen:No such file or directory:../crypto/bio/bss_file.c:69:fopen('C1','rb')
140645939307840:error:20060080:BIO routines:BIO_new_file:no such file:../crypto/bio/bss_file.c:76:
# ^C
# openssl enc -aes-128-cbc -e -in P1 -out C1
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.
# openssl enc -aes-128-cbc -d -nopad -in C1 -out P1_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
# xxd P1_new
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 12345.....
```

Figure 3 : creating a 5 bytes file through the terminal

- **00000000:** indicates the offset or position of the bytes in the file.
- **3132 3334 35** represents the ASCII characters for "12345", which is the content of the original file "P1" before encryption.
- **0b 0b 0b 0b 0b 0b 0b 0b** represents the padding added during the encryption process. Each "0b" corresponds to the hexadecimal value 0x0B, which is a non-printable control character used for padding.
- the padding is necessary to ensure the input data meets the required block size for the encryption algorithm. In this case, the AES-128-CBC algorithm used a block size of 16 bytes, so the padding was added to reach the full block size.

b) file 10 bytes

```
# echo -n "1234567890" > P2
# openssl enc -aes-128-cbc -e -in P2 -out C2
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
Verify failure
bad password read
139772101162304:error:2807106B:UI routines:UI_process:processing error:../crypto/ui/ui_lib.c:545:while reading strings
# openssl enc -aes-128-cbc -e -in P2 -out C2
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.
# openssl enc -aes-128-cbc -d -nopad -in C2 -out P2_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
# xxd P2_new
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890.....
■
```

Figure 4: : creating a 10 bytes file through the terminal

- **00000000:** indicates the offset or position of the bytes in the file.
- **3132 3334 3536 3738 3930** represents the ASCII characters for "1234567890", which is the content of the original file "P2" before encryption.
- **0606 0606 0606 0606** represents the padding added during the encryption process. Each "06" corresponds to the hexadecimal value 0x06, which is a non-printable control character used for padding.

Based on this output, we can conclude that during the encryption process, 6 bytes of padding consisting of the value 0x06 were added to the original file "P2" before encryption.

The padding is necessary to ensure the input data meets the required block size for the encryption algorithm. In this case, the AES-128-CBC algorithm used a block size of 16 bytes, so the padding was added to reach the full block size.

c) file 16 bytes

```
# echo -n "1234567890123456" > P3
# openssl enc -aes-128-cbc -e -in P3 -out C3
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.
# openssl enc -aes-128-cbc -d -nopad -in C3 -out P3_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
# xxd P3_new
00000000: 3132 3334 3536 3738 3930 3132 3334 3536  1234567890123456
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  .....
```

Figure 5 : : creating a 16 bytes file through the terminal

- **00000000:** indicates the offset or position of the bytes in the file.
- **3132 3334 3536 3738 3930 3132 3334 3536** represents the ASCII characters for "1234567890123456", which is the content of the original file "P3" before encryption.
- Since the original file size is equal to the block size (16 bytes), there is no need for additional padding.

3. Task 2 Padding Oracle Attack (Level 1)

part 1: The Oracle Setup

when we typed the command **nc 10.9.0.80 5000** we discovered that this command never existed so we needed to install it and these are the commands we typed to be able to interact with the oracle since the port 5000 was not listening, we tried pinging it and got no response so we had to install some commands packages and to edit over the compose file that was listed in the zip file:

```
Test-NetConnection 10.9.0.80 -Port 5000
```

```
apt-get update -y
```

```
apt-get install netcat -y
```

```
apt-get install iputils-ping -y
```

```
ping 10.9.0.80
```

```
PING 10.9.0.80 (10.9.0.80) 56(84) bytes of data.
```

```
not responding --> cannot see it
```

```
# ifconfig --> not on the same network
```

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
```

```
    inet 172.18.0.2 netmask 255.255.0.0 broadcast 172.18.255.255
```

```
    ether 02:42:ac:12:00:02 txqueuelen 0 (Ethernet)
```

```
    RX packets 6601 bytes 9344236 (8.9 MiB)
```

```
    RX errors 0 dropped 0 overruns 0 frame 0
```

```
    TX packets 3276 bytes 220251 (215.0 KiB)
```

```
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
```

```
    inet 127.0.0.1 netmask 255.0.0.0
```

```
    loop txqueuelen 1000 (Local Loopback)
```

```
    RX packets 28 bytes 2514 (2.4 KiB)
```

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 28 bytes 2514 (2.4 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

we have fixed the compose-dev file to connect to the same network and grant it the ip of 10.9.0.81 and now we have restarted the env, reinstalled the needed packages like netcat, ping and tested the ping again

ifconfig

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500

inet 10.9.0.81 netmask 255.255.255.0 broadcast 10.9.0.255

ether 02:42:0a:09:00:51 txqueuelen 0 (Ethernet)

RX packets 6591 bytes 9343786 (8.9 MiB)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 3284 bytes 219171 (214.0 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536

inet 127.0.0.1 netmask 255.0.0.0

loop txqueuelen 1000 (Local Loopback)

RX packets 28 bytes 2514 (2.4 KiB)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 28 bytes 2514 (2.4 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ping 10.9.0.80

PING 10.9.0.80 (10.9.0.80) 56(84) bytes of data.

64 bytes from 10.9.0.80: icmp_seq=1 ttl=64 time=0.132 ms

64 bytes from 10.9.0.80: icmp_seq=2 ttl=64 time=0.063 ms

64 bytes from 10.9.0.80: icmp_seq=3 ttl=64 time=0.090 ms

^C

--- 10.9.0.80 ping statistics ---

3 packets transmitted, 3 received, 0% packet loss, time 2080ms

rtt min/avg/max/mdev = 0.063/0.095/0.132/0.028 ms

###by doing all the steps above now this command works, the updated compose-

dev.yaml will be appended in the last pages

```
/*
```

```
services:
```

```
  app:
```

```
    entrypoint:
```

```
      - sleep
```

```
      - infinity
```

```
    image: docker/dev-environments-default:stable-1
```

```
    init: true
```

```
    volumes:
```

```
      - type: bind
```

```
        source: /var/run/docker.sock
```

```
        target: /var/run/docker.sock
```

```
networks:
```

```
  net-10.9.0.0:
```

```
    ipv4_address: 10.9.0.81
```

```
networks:
```

```
  net-10.9.0.0:
```

```
    name: net-10.9.0.0
```

```
    ipam:
```

```
      config:
```

```
        - subnet: 10.9.0.0/24
```

```
*/
```

Finally, after doing the needed installations and editing over the docker file we were able to run this command in the docker terminal and got the response we needed:

```
# netcat 10.9.0.80 5000
01020304050607080102030405060708a9b2554b0944118061212098f2f238cd779ea0aae3d9d020f3677bfc3cda9ce
```

*Netcat works just like nc

```
C: > Users > Lenovo > Downloads > Labsetup (1) > Labsetup > ! compose-dev.yaml
1  services:
2      app:
3          entrypoint:
4              - sleep
5              - infinity
6          image: docker/dev-environments-default:stable-1
7          init: true
8          volumes:
9              - type: bind
10                 source: /var/run/docker.sock
11                 target: /var/run/docker.sock
12
13         networks:
14             net-10.9.0.0:
15                 ipv4_address: 10.9.0.81
16
17     networks:
18         net-10.9.0.0:
19             name: net-10.9.0.0
20             ipam:
21                 config:
22                     - subnet: 10.9.0.0/24
23
24
```

Figure 6: setting the docker compose file for task2

Part 2- guessing the message:

To solve this task, we can utilize the padding oracle attack to recover the plaintext

message one block at a time. The padding oracle attack takes advantage of the fact that

the oracle will tell us whether the padding is valid or not.

PKCS#5 padding adds padding bytes to the plaintext to ensure that the total length of the plaintext is a multiple of the block size (16 bytes for AES-CBC). The value of each padding byte is set to the number of padding bytes added.

Here's the step-by-step approach for recovering the secret message:

1. We Connected to the suitable port in the oracle using the provided command:

netcat 10.9.0.80 5000.

```
# netcat 10.9.0.80 5000
01020304050607080102030405060708a9b2554b0944118061212098f2f238cd779ea0aae3d9d020f3677bfc3cda9ce
```

2. And we gave the system the IV and cipher text that we got to see its response and it really said that its valid

```
01020304050607080102030405060708a9b2554b0944118061212098f2f238cd779ea0aae3d9d020f3677bfc3cda9ce
Valid
```

it indicates that the padding is valid for the last byte of the plaintext block.

3. Then we changed the last block of the IV from 08 to 12 and concatenated it to the cipher text and it gave me valid too

```
# netcat 10.9.0.80 5000
01020304050607080102030405060708a9b2554b0944118061212098f2f238cd779ea0aae3d9d020f3677bfc3cda9ce
01020304050607080102030405060712a9b2554b0944118061212098f2f238cd779ea0aae3d9d020f3677bfc3cda9ce
Valid
```

4. Here are the commands used labeled in yellow I wrote in the terminal using the docker in order to install python compiler hence opening the manual_attak file

```
# apt install software-properties-common -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done

The following additional packages will be installed:
  dbus dmsetup gir1.2-glib-2.0 gir1.2-packagekit-glib-1.0 iso-codes libapparmor1 libappstream4 libargon2-1 libcryptsetup12 libdbus-1-3 libdevmapper1.02.1 libdw1
  libgirepository1.0-1 libgl100-0 libgl100-bin libgl100-data libgststreamer1-0-0 libip4tc2 libjson-c5 libkmod2 libldb0 libnss-systemd libpackagekit-glib2-18
  libpam-systemd libpolkit-agent-1-0 libpolkit-gobject-1-0 libstemmer0d libsystemd0 libxml2 libyaml-0-2 packagekit packagekit-tools policykit-1 python-apt-common python3-apt
  python3-dbus python3-distutils python3-gi python3-pycurl python3-software-properties sensible-utils shared-mime-info systemd systemd-sysv systemd-timesyncd ucf
  unattended-upgrades xdg-user-dirs xz-utils

Suggested packages:
  default-dbus-session-bus | dbus-session-bus isoquery gststreamer1.0-tools appstream python3-dbg python-apt-doc python-dbus-doc python3-dbus-dbg libcurl4-gnutls-dev
  python-pycurl-doc python3-pycurl-dbg systemd-container bsd-mailx default-ntp | ntp transport-agent needrestart powertop base

The following NEW packages will be installed:
  dbus dmsetup gir1.2-glib-2.0 gir1.2-packagekit-glib-1.0 iso-codes libapparmor1 libappstream4 libargon2-1 libcryptsetup12 libdbus-1-3 libdevmapper1.02.1 libdw1
  libgirepository1.0-1 libgl100-0 libgl100-bin libgl100-data libgststreamer1-0-0 libip4tc2 libjson-c5 libkmod2 libldb0 libnss-systemd libpackagekit-glib2-18
  libpam-systemd libpolkit-agent-1-0 libpolkit-gobject-1-0 libstemmer0d libsystemd0 libxml2 libyaml-0-2 packagekit packagekit-tools policykit-1 python-apt-common python3-apt
  python3-dbus python3-distutils python3-gi python3-pycurl python3-software-properties sensible-utils shared-mime-info software-properties-common systemd systemd-sysv systemd-timesyncd
  ucf unattended-upgrades xdg-user-dirs xz-utils

The following packages will be upgraded:
  libsystemd0
  1 upgraded, 49 newly installed, 0 to remove and 34 not upgraded.
Need to get 19.1 MB of archives.
After this operation, 79.8 MB of additional disk space will be used.
Get:1 http://deb.debian.org/debian bullseye/main amd64 libsystemd0 amd64 247.3-7+deb11u2 [376 kB]
Get:2 http://deb.debian.org/debian bullseye/main amd64 libapparmor1 amd64 2.13.6-10 [99.3 kB]
Get:3 http://deb.debian.org/debian bullseye/main amd64 libargon2-1 amd64 0-20171227-0.2 [19.6 kB]
Get:4 http://deb.debian.org/debian bullseye/main amd64 dmsetup amd64 2:1.02.175-2.1 [92.1 kB]
Get:5 http://deb.debian.org/debian bullseye/main amd64 libdevmapper1.02.1 amd64 2:1.02.175-2.1 [143 kB]
Get:6 http://deb.debian.org/debian bullseye/main amd64 libjson-c5 amd64 0.15-2 [42.8 kB]
Get:7 http://deb.debian.org/debian bullseye/main amd64 libcryptsetup12 amd64 2:2.3.7-1+deb11u1 [248 kB]
Get:8 http://deb.debian.org/debian bullseye/main amd64 libip4tc2 amd64 1.8.7-1 [34.6 kB]
```

```
Get:6 http://deb.debian.org/debian bullseye/main amd64 libjson-c5 amd64 0.15-2 [42.8 kB]
Get:7 http://deb.debian.org/debian bullseye/main amd64 libcryptsetup12 amd64 2:2.3.7-1+deb11u1 [248 kB]
Get:8 http://deb.debian.org/debian bullseye/main amd64 libip4tc2 amd64 1.8.7-1 [34.6 kB]
Get:9 http://deb.debian.org/debian bullseye/main amd64 libkmod2 amd64 28-1 [55.6 kB]
Get:10 http://deb.debian.org/debian bullseye/main amd64 systemd amd64 247.3-7+deb11u2 [4501 kB]
Get:11 http://deb.debian.org/debian bullseye/main amd64 libsystemd-sysv amd64 247.3-7+deb11u2 [113 kB]
Get:12 http://deb.debian.org/debian bullseye/main amd64 libdbus-1-3 amd64 1.12.24-0+deb11u1 [222 kB]
Get:13 http://deb.debian.org/debian bullseye/main amd64 dbus amd64 1.12.24-0+deb11u1 [243 kB]
Get:14 http://deb.debian.org/debian bullseye/main amd64 sensible-utils all 0.0.14 [14.8 kB]
Get:15 http://deb.debian.org/debian bullseye/main amd64 libnss-systemd amd64 247.3-7+deb11u2 [199 kB]
Get:16 http://deb.debian.org/debian bullseye/main amd64 libpam-systemd amd64 247.3-7+deb11u2 [283 kB]
Get:17 http://deb.debian.org/debian bullseye/main amd64 systemd-timesyncd amd64 247.3-7+deb11u2 [131 kB]
Get:18 http://deb.debian.org/debian bullseye/main amd64 ucf all 3.0043 [74.0 kB]
Get:19 http://deb.debian.org/debian bullseye/main amd64 xz-utils amd64 5.2.5-2-1+deb11u1 [220 kB]
Get:20 http://deb.debian.org/debian bullseye/main amd64 libgl100-0 amd64 2.66.8-1 [1370 kB]
Get:21 http://deb.debian.org/debian bullseye/main amd64 libgirepository1.0-1 amd64 1.66.1-1+b1 [96.7 kB]
Get:22 http://deb.debian.org/debian bullseye/main amd64 gir1.2-glib-2.0 amd64 1.66.1-1+b1 [151 kB]
Get:23 http://deb.debian.org/debian bullseye/main amd64 libpackagekit-glib2-18 amd64 1.2.2-2 [124 kB]
Get:24 http://deb.debian.org/debian bullseye/main amd64 gir1.2-packagekit-glib-1.0 amd64 1.2.2-2 [36.8 kB]
Get:25 http://deb.debian.org/debian bullseye/main amd64 iso-codes all 4.6.0-1 [2824 kB]
Get:26 http://deb.debian.org/debian bullseye/main amd64 libldb0 amd64 0.9.24-1 [45.0 kB]
Get:27 http://deb.debian.org/debian bullseye/main amd64 libstemmer0d amd64 2.1.0-1 [119 kB]
Get:28 http://deb.debian.org/debian bullseye/main amd64 libxml2 amd64 2.9.10+dfsg-6.7+deb11u4 [693 kB]
Get:29 http://deb.debian.org/debian bullseye/main amd64 libyaml-0-2 amd64 0.2.2-1 [49.6 kB]
Get:30 http://deb.debian.org/debian bullseye/main amd64 libappstream4 amd64 0.14.4-1 [172 kB]
Get:31 http://deb.debian.org/debian bullseye/main amd64 libdw1 amd64 0.183-1 [234 kB]
Get:32 http://deb.debian.org/debian bullseye/main amd64 libgl100-0-data all 2.66.8-1 [1164 kB]
Get:33 http://deb.debian.org/debian bullseye/main amd64 libgl100-bin amd64 2.66.8-1 [141 kB]
Get:34 http://deb.debian.org/debian bullseye/main amd64 libgststreamer1-0-0 amd64 1.18.4-2.1 [2230 kB]
Get:35 http://deb.debian.org/debian bullseye/main amd64 libpolkit-gobject-1-0 amd64 0.105-31+deb11u1 [48.5 kB]
Get:36 http://deb.debian.org/debian bullseye/main amd64 libpolkit-agent-1-0 amd64 0.105-31+deb11u1 [28.1 kB]
Get:37 http://deb.debian.org/debian bullseye/main amd64 policykit-1 amd64 0.105-31+deb11u1 [96.7 kB]
```

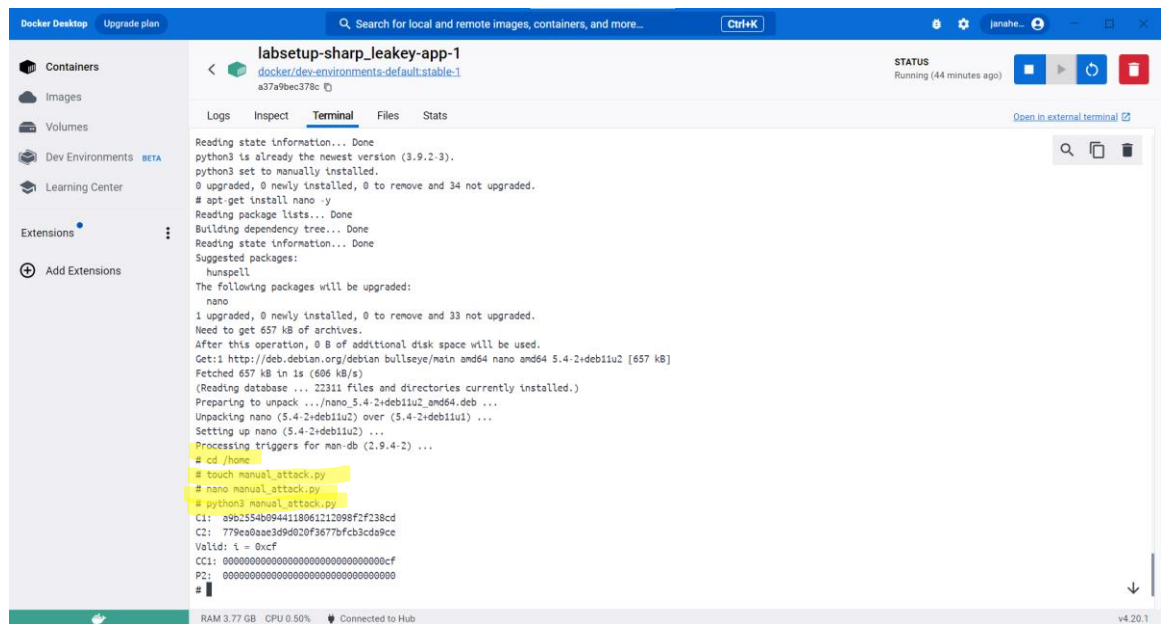



Figure 8 : running the manual_attack file after installing python3

By now we have successfully opened and run manual_attack.py through the terminal and here is the output of it that includes two block ciphers

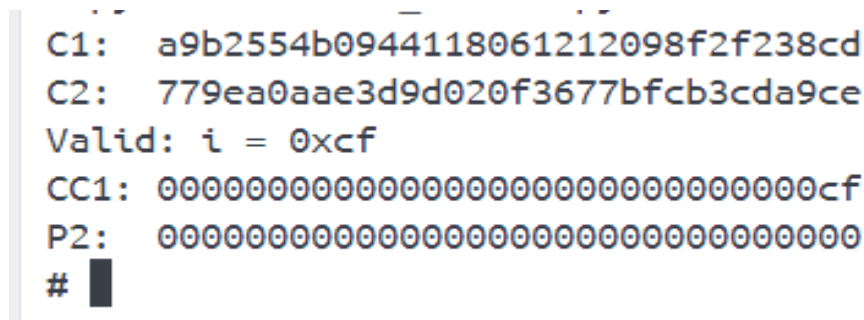
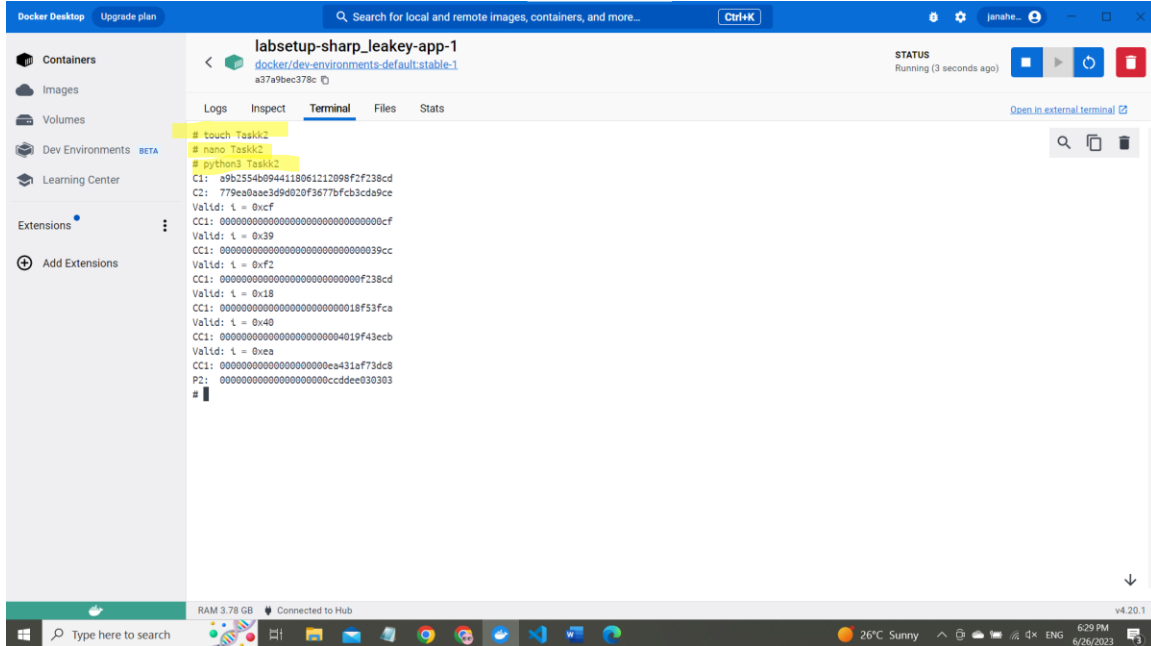


Figure 9 : zoomed in picture of the output

*This is a zoomed-in picture of the output that we got from running the python file in the docker app terminal.

Now we need to find the other blocks (total 6 blocks as required). So, first thing we modified the file attached in the zip file (manual_attack.py) and we want to run it through the terminal.

So, we used these terminals in yellow to find them



```
labsetup-sharp_leakey-app-1
docker/dev-environments-default:stable-1
a37a9bec378e

# touch Taskk2
# nano Taskk2
# python3 Taskk2
C1: 89b2554b9944118061212098f2f238cd
C1: 779ec8a9c3d9d020f3677bfcb3cda9ce
Valid: t = 0xcdf
CC1: 000000000000000000000000000000cf
Valid: t = 0x39
CC1: 00000000000000000000000000000039cc
Valid: t = 0xf2
CC1: 000000000000000000000000000000f238cd
Valid: t = 0x18
CC1: 00000000000000000000000000000018f53fca
Valid: t = 0x4b
CC1: 0000000000000000000000000000004b19f43ecb
Valid: t = 0xea
CC1: 000000000000000000000000000000ea431ef73dc8
P2: 000000000000000000000000000000ccdee030303
#
```

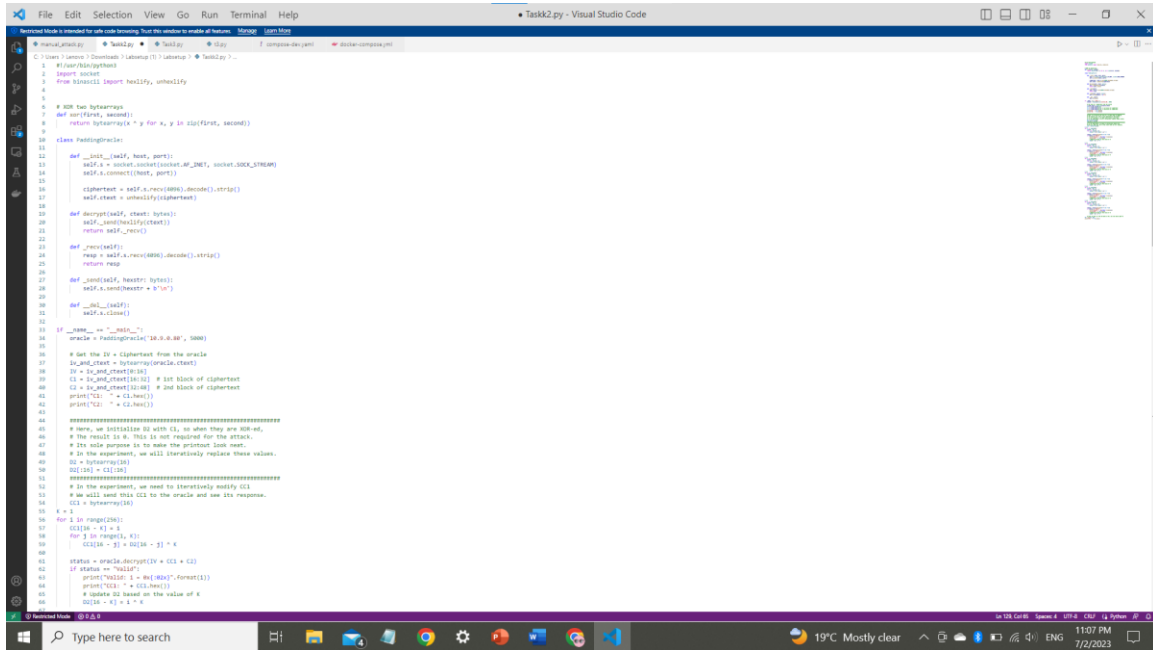
Figure 10 : running Task2.py file

Taskk2.py is the name of the python file that we modified to get the other blocks of the message

So, the blocks that we got are named in cc1 and we did the same thing for each k from k=1 to k =6 and they were printed out.

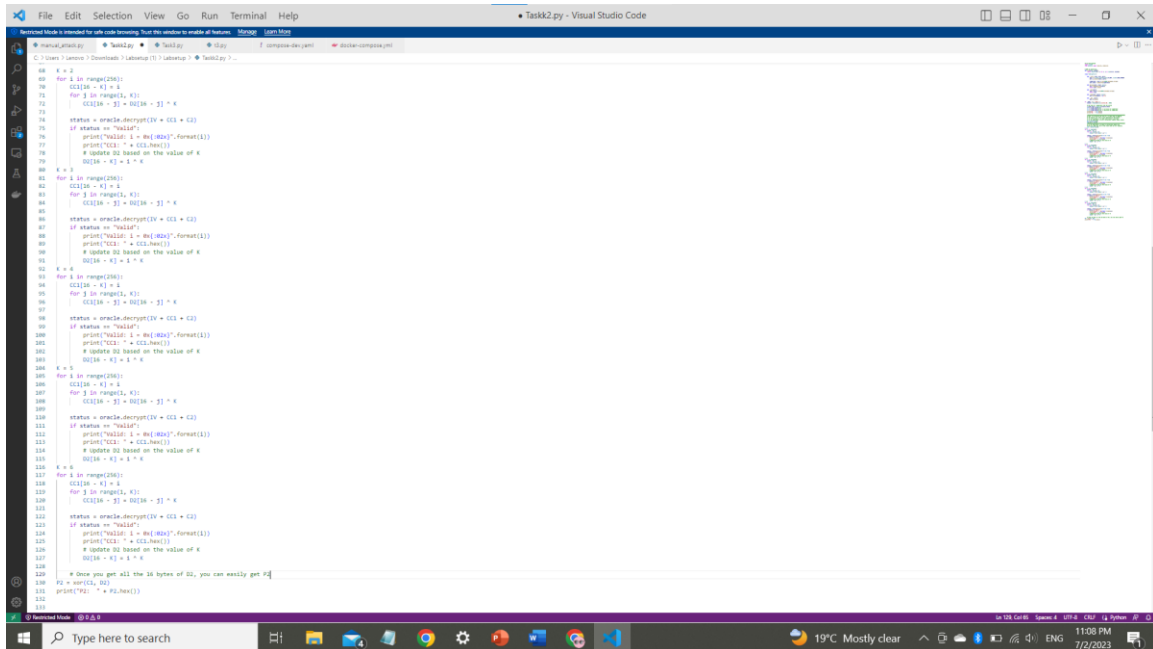
And this was the effective blocks of the code for task2 which will be appended in the appendix part:

Here is a screenshot of Taskk2.py file it will also be appended in the end of the report

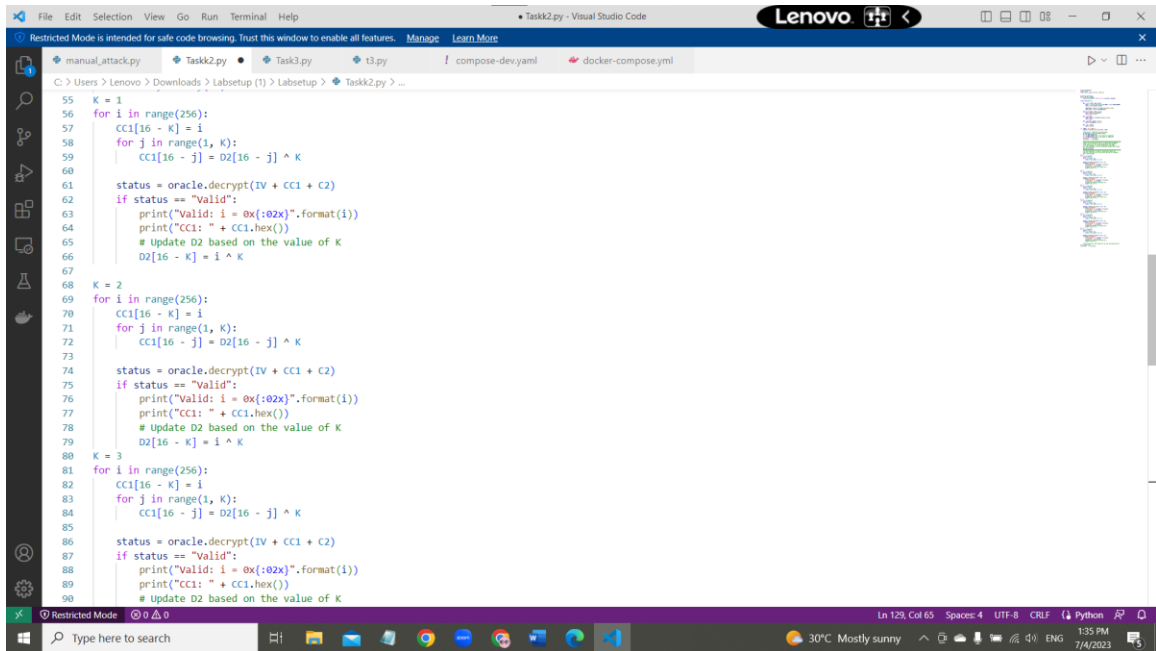


The screenshot shows the Visual Studio Code editor with the file 'Taskk2.py' open. The code is a Python script for a cryptographic challenge. It starts with a comment 'Reverse Mode -> intended for safe code auditing. That this enables to enable of features.' and includes imports for 'socket', 'struct', 'sys', and 'binascii'. The script defines a 'PadingOracle' class with methods for connecting to a host, sending and receiving data, and decoding/encoding. It then defines a 'PadingOracle' class with methods for connecting to a host, sending and receiving data, and decoding/encoding. The main part of the script is a loop that sends a series of 'A' characters to the oracle and receives a response. It then uses a binary search algorithm to find the correct value for 'x' by comparing the response to a known value. The script ends with a comment 'Now you get all the 16 bytes of x, you can easily get P0'.

Figure 11 : screenshot of Task2.py on my device

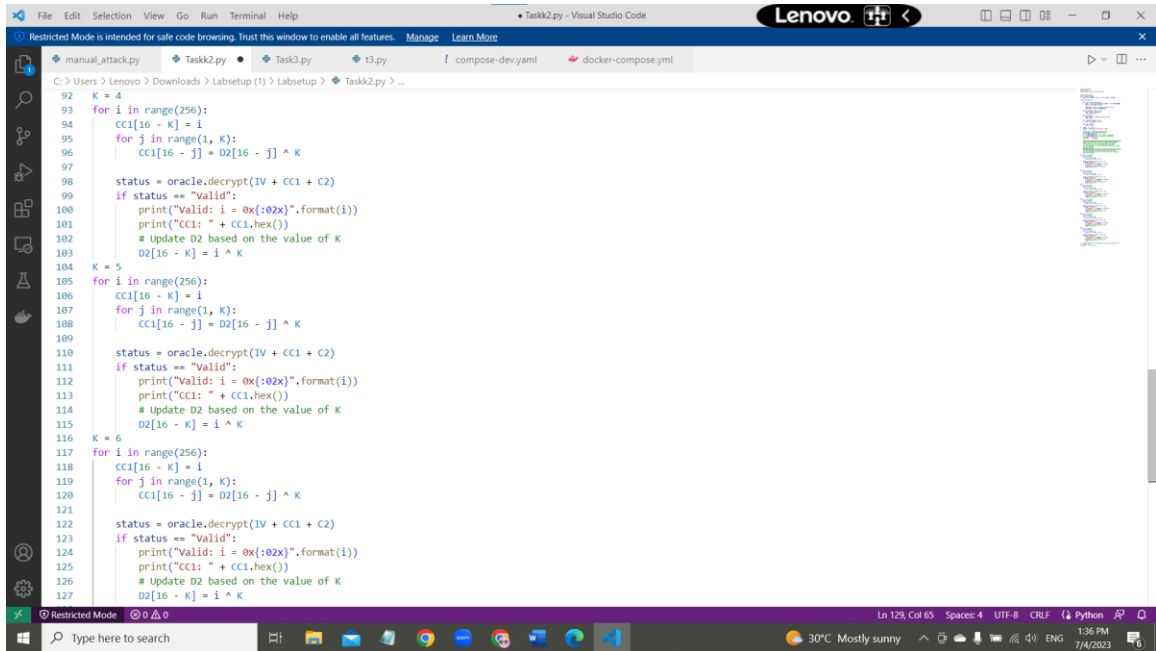


This screenshot shows the same Visual Studio Code editor with the 'Taskk2.py' file. The code is identical to the one in the previous screenshot, showing the same imports, class definitions, and the main loop for the cryptographic challenge. The script is designed to interact with a remote oracle and perform a binary search to determine the value of 'x'.



```
55 K = 1
56 for i in range(256):
57     CC1[16 - K] = i
58     for j in range(1, K):
59         CC1[16 - j] = D2[16 - j] ^ K
60
61     status = oracle.decrypt(IV + CC1 + C2)
62     if status == "Valid":
63         print("Valid: i = 0x{:02x}".format(i))
64         print("CC1: " + CC1.hex())
65         # Update D2 based on the value of K
66         D2[16 - K] = i ^ K
67
68 K = 2
69 for i in range(256):
70     CC1[16 - K] = i
71     for j in range(1, K):
72         CC1[16 - j] = D2[16 - j] ^ K
73
74     status = oracle.decrypt(IV + CC1 + C2)
75     if status == "Valid":
76         print("Valid: i = 0x{:02x}".format(i))
77         print("CC1: " + CC1.hex())
78         # Update D2 based on the value of K
79         D2[16 - K] = i ^ K
80
81 K = 3
82 for i in range(256):
83     CC1[16 - K] = i
84     for j in range(1, K):
85         CC1[16 - j] = D2[16 - j] ^ K
86
87     status = oracle.decrypt(IV + CC1 + C2)
88     if status == "Valid":
89         print("Valid: i = 0x{:02x}".format(i))
90         print("CC1: " + CC1.hex())
91         # Update D2 based on the value of K
```

Figure 12 : first 3 blocks from k=1 to k=3



```
92 K = 4
93 for i in range(256):
94     CC1[16 - K] = i
95     for j in range(1, K):
96         CC1[16 - j] = D2[16 - j] ^ K
97
98     status = oracle.decrypt(IV + CC1 + C2)
99     if status == "Valid":
100         print("Valid: i = 0x{:02x}".format(i))
101         print("CC1: " + CC1.hex())
102         # Update D2 based on the value of K
103         D2[16 - K] = i ^ K
104
105 K = 5
106 for i in range(256):
107     CC1[16 - K] = i
108     for j in range(1, K):
109         CC1[16 - j] = D2[16 - j] ^ K
110
111     status = oracle.decrypt(IV + CC1 + C2)
112     if status == "Valid":
113         print("Valid: i = 0x{:02x}".format(i))
114         print("CC1: " + CC1.hex())
115         # Update D2 based on the value of K
116         D2[16 - K] = i ^ K
117
118 K = 6
119 for i in range(256):
120     CC1[16 - K] = i
121     for j in range(1, K):
122         CC1[16 - j] = D2[16 - j] ^ K
123
124     status = oracle.decrypt(IV + CC1 + C2)
125     if status == "Valid":
126         print("Valid: i = 0x{:02x}".format(i))
127         print("CC1: " + CC1.hex())
128         # Update D2 based on the value of K
129         D2[16 - K] = i ^ K
```

Figure 13 :second 3 blocks from k=4 to k=6

Desired blocks:

```
C1: a9b2554b0944118061212098f2f238cd
C2: 779ea0aae3d9d020f3677bfc3cda9ce
Valid: i = 0xcf
CC1: 000000000000000000000000000000cf
Valid: i = 0x39
CC1: 00000000000000000000000000000039cc
Valid: i = 0xf2
CC1: 000000000000000000000000000000f238cd
Valid: i = 0x18
CC1: 00000000000000000000000000000018f53fca
Valid: i = 0x40
CC1: 0000000000000000000000000000004019f43ecb
Valid: i = 0xea
CC1: 000000000000000000000000000000ea431af73dc8
P2: 000000000000000000000000000000ccdde030303
#
```

Figure 14 : desired blocks 6 blocks of D2 manually runned

Explanation:

Task 2 involved the execution of a padding oracle attack, exploiting systems that verify the validity of padding during the decryption process. The primary objective was to decrypt a confidential message without possessing the knowledge of the encryption key. The Advanced Encryption Standard with Cipher Block Chaining (AES-CBC) algorithm was employed for encryption.

A dedicated padding oracle was established on port 5000, providing a ciphertext of the encrypted secret message, which encompassed an Initialization Vector (IV) and multiple blocks.

The ciphertext was retrieved from the padding oracle and stored in two distinct 16-byte arrays, namely C1 and C2.

To exploit the vulnerability of the padding oracle, the attack involved the transmission of modified versions of C1 (referred to as CC1) alongside C2, allowing for the observation of for each k from 1 to 6, each byte of the intermediate array D2 was gradually decrypted. This intermediate array represented the output of the block cipher prior to the XOR operation with the previous ciphertext block (or IV for the initial block). Once D2 was successfully determined, the corresponding plaintext block P2 was derived by performing an XOR operation between C1 and D2, resulting in $P2 = C1 \oplus D2$.

To perform the padding oracle attack for different values of K (ranging from 1 to 6):

- We Initialized the CC1 (modified ciphertext block) bytearray with all zeros.
- Iterated over the range of 256 (all possible byte values) for each K .
- Set the K th byte of CC1 to the current iteration value.
- For each byte position from 1 to K , XOR the corresponding byte of D2 with K and store it in CC1.
- the modified ciphertext (IV + CC1 + C2) was sent to the oracle for decryption.
- Checked if the oracle's response is "Valid."
 - If valid, printed the value of i (the current iteration byte value) and the hexadecimal representation of CC1.

- Updated the Kth byte of D2 based on the value of K by XORing it with K.

Then After obtaining all 16 bytes of D2, P2 (the plaintext of the second block) was calculated by XORing C1 with D2. And printed the hexadecimal representation of P2.

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

First part: is to make the attack automatically done not manually so I have edited the manual _attack.py to be Task3-automatedtask2 and here **its contents**

So, we wrote this in the docker terminal to open and run the file.

```

Valid for block=3 K=12: t = 0xdf
CC: 00000000f46a9d311823ee9706eb00
Valid for block=3 K=13: t = 0xff
CC: 000000ffde47a1d220833fe8716fbc01
Valid for block=3 K=14: t = 0x99
CC: 000099fcd444a2d123803ceb726cbf02
Valid for block=3 K=15: t = 0x55
CC: 005598f6dc45a3d022313dea736db03
Valid for block=3 K=16: t = 0x42
CC: 424a87e2c35abccf3d9e22f96c72a11c
P1: 285e5f5e29285e5f5e29205468652053
P2: 454544204c61627732061726520677265
P3: 61742120285e5f5e29285e5f5e290202
# touch Task3-automatedtask2.py
# nano Task3-automatedtask2.py
# python3 Task3-automatedtask2.py
C1: a9b2354b0944118061212098f2f238cd
C2: 779ea0aee3d9d020f3677bfcb3cd89ce
Valid: t = 0xcf
CC1: 0000000000000000000000000000cf
Valid: t = 0x39
CC1: 000000000000000000000000000039cc
Valid: t = 0xf2
CC1: 00000000000000000000000000f238cd
Valid: t = 0x18
CC1: 0000000000000000000000000018f53fca
Valid: t = 0x40
CC1: 000000000000000000000000004019f43ecb
Valid: t = 0xea
CC1: 000000000000000000000000ea431af73dc8
P2: 000000000000000000000000ccdde030303
#

```

Figure 15 : running Task3-automatedtask2 through the docker terminal

Here is a zoomed in picture of the output:


```

# touch Task3-automatedtask2.py
# nano Task3-automatedtask2.py
# python3 Task3-automatedtask2.py
C1: a9b2554b0944118061212098f2f238cd
C2: 779ea0aae3d9d020f3677bfc3cda9ce
Valid: i = 0xcf
CC1: 000000000000000000000000000000cf
Valid: i = 0x39
CC1: 00000000000000000000000000000039cc
Valid: i = 0xf2
CC1: 000000000000000000000000000000f238cd
Valid: i = 0x18
CC1: 00000000000000000000000000000018f53fca
Valid: i = 0x40
CC1: 0000000000000000000000000000004019f43ecb
Valid: i = 0xea
CC1: 000000000000000000000000000000ea431af73dc8
P2: 000000000000000000000000000000ccdde030303
# █

```

Figure 16 : 6 bytes of D2 automatically runned

We can see that the same output was gained using the automatically looped and here is a snapshot of the file of Task3-automatedtask2 and the code will be appended at the end of the report which is completely as the past code but instead of writing each k manually we wrote it as a loop that iterate through k form 1 to 6 without the need of writing each case alone.

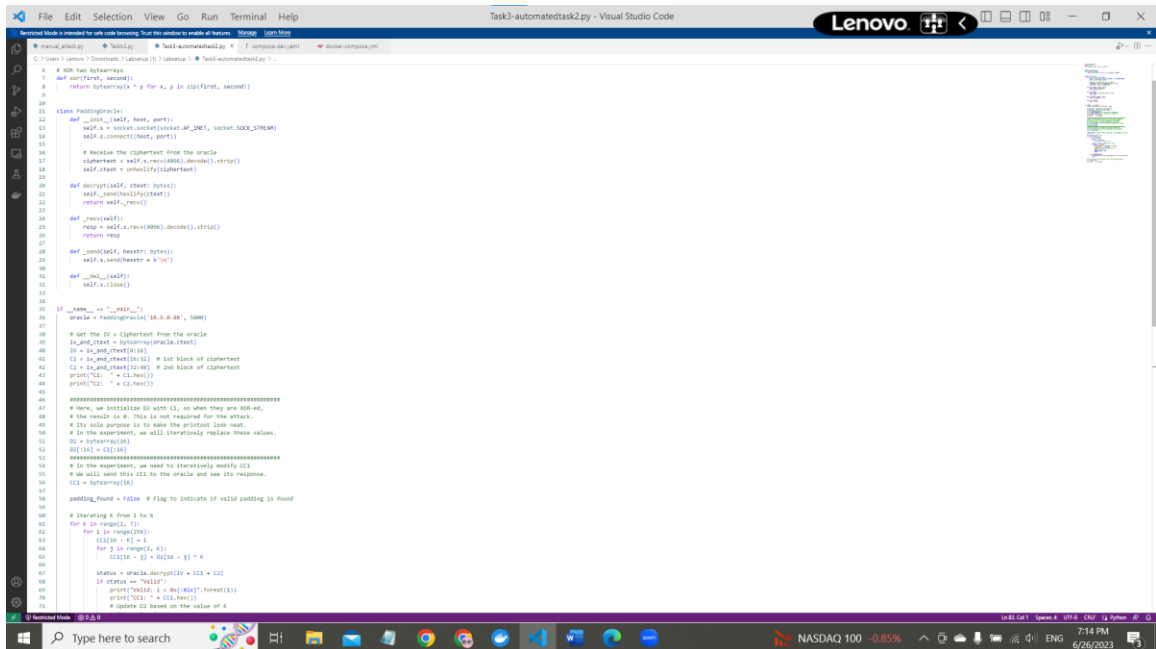


Figure 17: Automated task2

Second part:

First thing we need to connect to port 6000 of the oracle servers

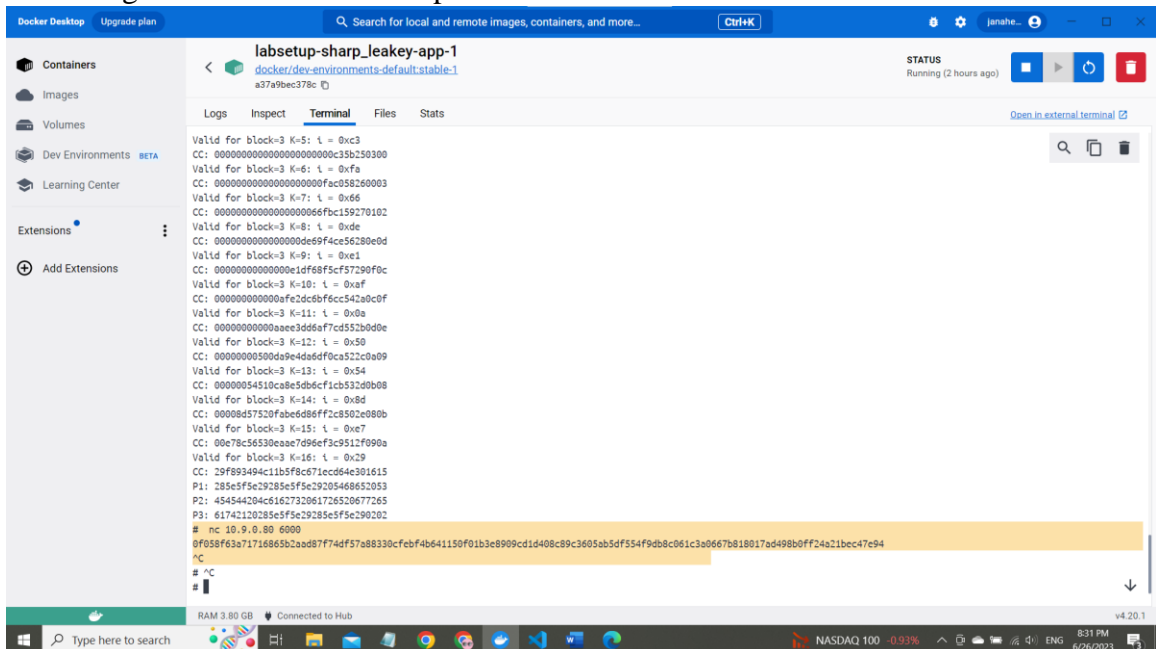
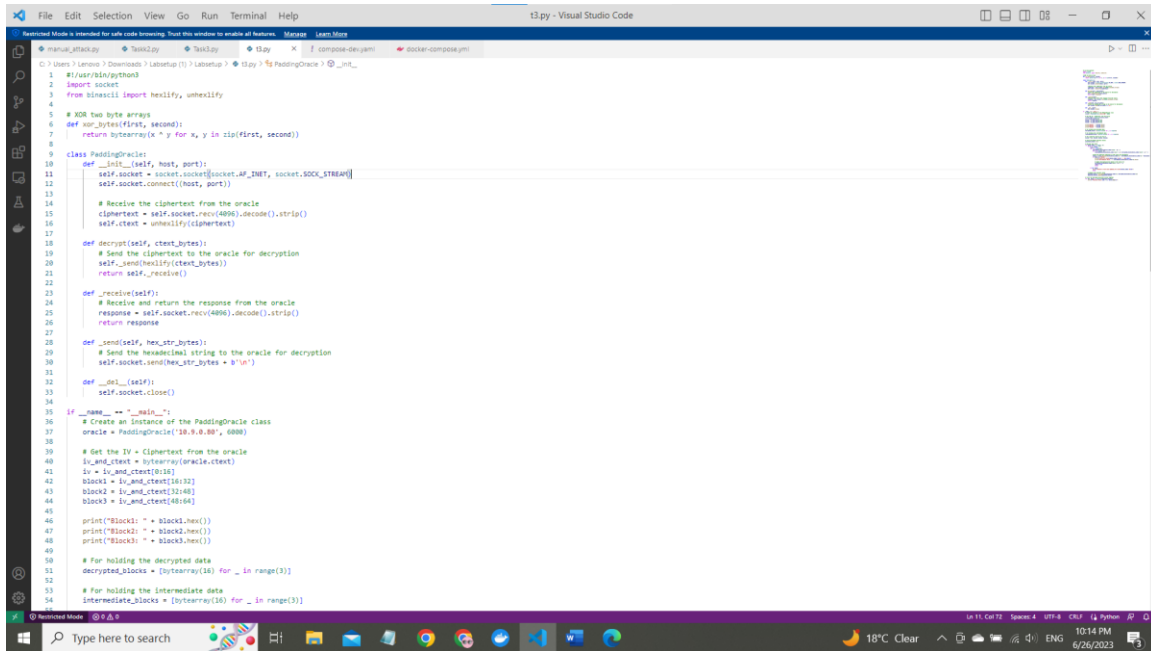


Figure 18 : connecting to port 6000 through the docker terminal

So, by running the nc 10.9.0.80 6000 now the oracle is listening and ready to be attacked to find out the message.

For task3 we have created the Task3.py file to find the message hidden inside and here is a snap of the code



```
1 #!/usr/bin/python3
2 import socket
3 from binascii import hexlify, unhexlify
4
5 # XOR two byte arrays
6 def xor_bytes(first, second):
7     return bytearray(x ^ y for x, y in zip(first, second))
8
9 class PaddingOracle:
10     def __init__(self, host, port):
11         self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12         self.socket.connect((host, port))
13
14         # Receive the ciphertext from the oracle
15         ciphertext = self.socket.recv(4096).decode().strip()
16         self.ciphertext = unhexlify(ciphertext)
17
18     def decrypt(self, ctext_bytes):
19         # Send the ciphertext to the oracle for decryption
20         self.send_hexlify(ctext_bytes)
21         return self.receive()
22
23     def _receive(self):
24         # Receive and return the response from the oracle
25         response = self.socket.recv(4096).decode().strip()
26         return response
27
28     def _send(self, hex_str_bytes):
29         # Send the hexstring to the oracle for decryption
30         self.socket.send(hex_str_bytes + '\n')
31
32     def __del__(self):
33         self.socket.close()
34
35 if __name__ == '__main__':
36     # Create an instance of the PaddingOracle class
37     oracle = PaddingOracle('10.9.0.88', 4080)
38
39     # Get the IV = ciphertext from the oracle
40     iv_and_ciphertext = bytearray(oracle.ciphertext)
41     iv = iv_and_ciphertext[0:16]
42     block1 = iv_and_ciphertext[16:32]
43     block2 = iv_and_ciphertext[32:48]
44     block3 = iv_and_ciphertext[48:64]
45
46     print("Block1: " + block1.hex())
47     print("Block2: " + block2.hex())
48     print("Block3: " + block3.hex())
49
50     # For holding the decrypted data
51     decrypted_blocks = [bytearray(16) for _ in range(3)]
52
53     # For holding the intermediate data
54     intermediate_blocks = [bytearray(16) for _ in range(3)]
55
56     # The ciphertext blocks and the IV
57     blocks = [iv, block1, block2, block3]
58
59     # Store decrypted plaintext blocks
60     plaintext_blocks = []
61
62     # Decrypt each block
63     for block_index in range(1, 4):
64         for k in range(1, 17):
65             found = False
66             for i in range(16):
67                 intermediate_blocks[block_index-1][16 - k] = i
68                 for j in range(1, k):
69                     intermediate_blocks[block_index-1][16 - j] = decrypted_blocks[block_index-1][16 - j] ^ k
70
71             # Send the modified ciphertext to the oracle for decryption
72             status = oracle.decrypt(blocks[block_index-1] + intermediate_blocks[block_index-1])
73             if status == "Valid":
74                 print("Valid for block: " + str(block_index) + " k: " + str(k) + " i: " + str(i))
75                 print("Intermediate Block: " + intermediate_blocks[block_index-1].hex())
76
77             # Update decrypted_blocks based on the value of k
78             decrypted_blocks[block_index-1][16 - k] = i ^ k
79             found = True
80             break
81
82         if not found:
83             print("Failed to find valid padding for block: " + str(block_index) + " k: " + str(k))
84             break
85
86     # Compute the plaintext block
87     plaintext_block = xor_bytes(blocks[block_index-1], decrypted_blocks[block_index-1])
88     plaintext_blocks.append(plaintext_block)
89
90     # Print the decrypted plaintext blocks together
91     for idx, block in enumerate(plaintext_blocks):
92         print("Plaintext Block (idx = " + str(idx) + "): " + block.hex())
```

Figure 19 : Task3.py file running in visual studio

Explanation:

The code establishes a connection with the padding oracle and receives the ciphertext, which consists of an initialization vector (IV) and three blocks: **block1**, **block2**, and **block3**. It then initializes data structures to store the decrypted blocks (**decrypted_blocks**) and intermediate blocks (**intermediate_blocks**).

The decryption process begins by iterating over each block. Within each block, there are nested loops that attempt different padding values (**k**) and byte values (**i**) to deduce the correct padding and recover the plaintext. The intermediate block is adjusted based on the padding and byte values, and the modified ciphertext is sent to the padding oracle for decryption.

If the padding oracle responds with "Valid," indicating that the padding is correct, the intermediate block is stored in **intermediate_blocks**. The decrypted byte for the current position (**k**) is determined by XORing the intermediate block with the padding value (**k**). In cases where the padding is invalid or no valid padding is found, the process is halted for the current block.

After decrypting each block, the plaintext blocks are computed by XORing the decrypted blocks with the corresponding ciphertext blocks. Finally, the decrypted plaintext blocks are printed.

This code showcases a practical application of padding oracle attacks, where the presence or absence of valid padding is leveraged to decrypt ciphertext. By following the provided steps, the code aims to recover the original plaintext from the given ciphertext.

The `if __name__ == "__main__":` condition ensures that the code block is only executed if the script is being run directly and not imported as a module.

An instance of the **PaddingOracle** class is created by providing the IP address ('10.9.0.80') and port (6000) as arguments to the constructor. This establishes a connection with the padding oracle.

The ciphertext (IV + Ciphertext) is received from the oracle, stored in the **ctext** attribute of the **oracle** instance, and then split into the IV and three blocks (**block1**, **block2**, **block3**).

The IV and ciphertext blocks are printed for reference.

Data structures are initialized to hold the decrypted blocks (**decrypted_blocks**) and intermediate blocks (**intermediate_blocks**). Each of these lists contains three bytearrays, with each bytearray having a length of 16 bytes.

The IV, ciphertext blocks, and intermediate blocks are stored in the **blocks** list, which will be used for decryption.

An empty list **plaintext_blocks** is created to store the decrypted plaintext blocks.

The decryption process begins with a nested loop structure. The outer loop iterates over the block indices (**block_index**) from 1 to 3.

Within each block, the inner loops iterate over possible padding lengths (**k**) from 1 to 16.

For each padding length (**k**), a loop iterates over possible byte values (**i**) from 0 to 255.

The intermediate block at position **16 - k** is set to the byte value **i**. Additionally, the intermediate block is computed for positions **16 - j** for each **j** from 1 to **k**.

The modified ciphertext (consisting of the preceding ciphertext block, the current intermediate block, and the next ciphertext block) is sent to the padding oracle for decryption.

If the padding oracle responds with "Valid," it means the padding is correct. The intermediate block is stored, and the decrypted byte for the current position (**k**) is computed by XORing the intermediate block with the padding value (**k**). The updated decrypted byte is stored in **decrypted_blocks**.

If no valid padding is found for a given **k**, the process is aborted for the current block, and a failure message is printed.

After all the padding lengths (**k**) have been processed for a block, the decrypted block is computed by XORing the corresponding ciphertext block with the decrypted block.

The decrypted block is appended to the **plaintext_blocks** list.

Once all three blocks have been decrypted, the code prints the decrypted plaintext blocks, displaying their respective indices and hexadecimal representations.

Conclusion of task3:

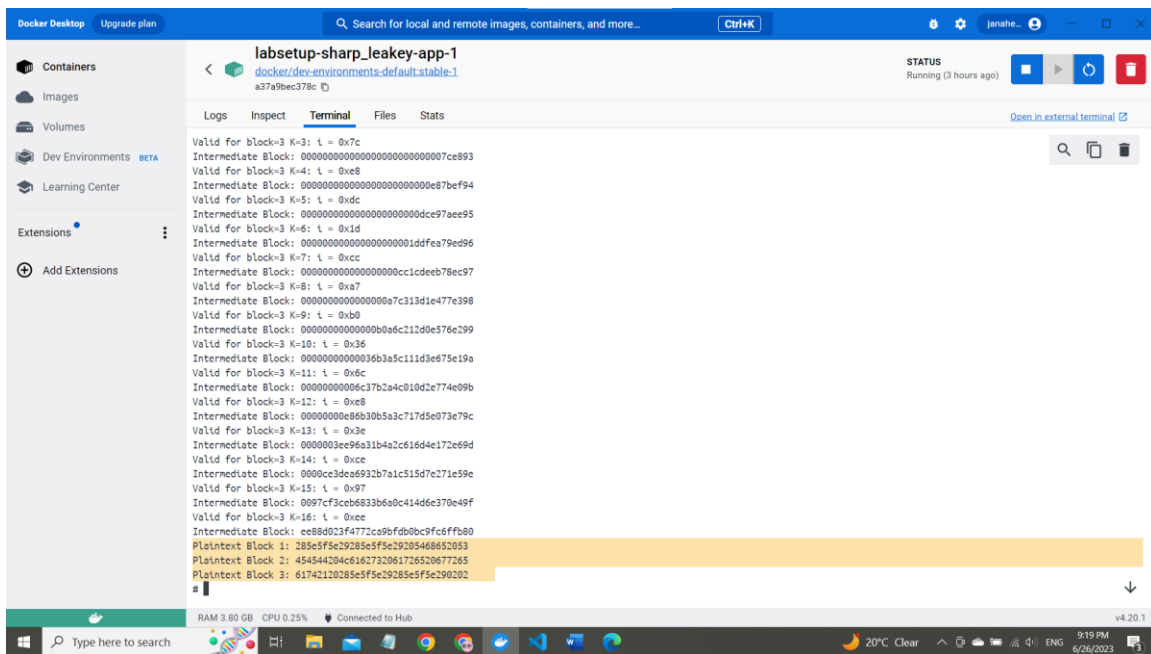
In task 3, the padding oracle attack procedure was automated and expanded to decrypt every block of the secret message. The goal was to use the padding oracle to obtain the whole plaintext without having access to the encryption key in advance. Again, the encryption technique was AES-CBC. On port 6000, a Level 2 padding Oracle server was built up to facilitate interaction during the automated attack. An IV and encrypted message blocks were included in the ciphertext that the server returned in response to requests. The ciphertext was first retrieved from the padding oracle. To hold the various

parts of the ciphertext, four 16-byte arrays with an IV and three ciphertext blocks (C1, C2, and C3) were built.

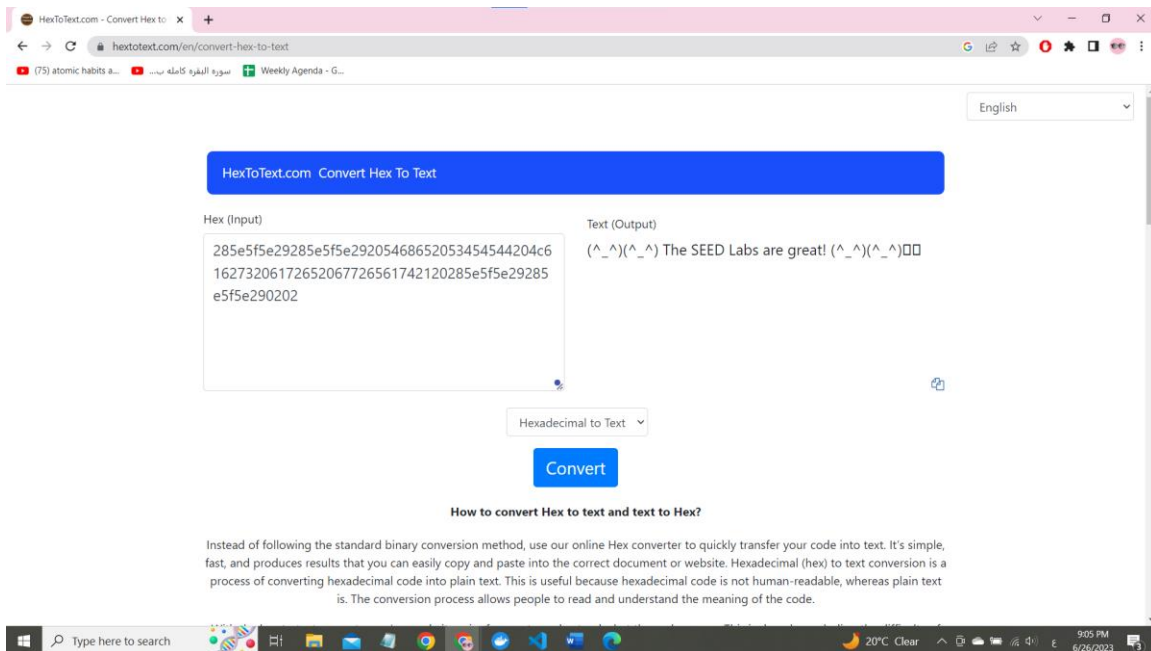
Arrays were initialized before the automated assault was launched, the intermediate data (CC) and decrypted data (D) were placed in the initialized arrays. The IV and the blocks of the ciphertext were allocated to the appropriate variables. Iterative block-by-block decryption was used in the attack.

One byte at a time was the focus of an iterative procedure for each block. In order to find out whether the padding was reliable, the plan was to alter CC1 in each round, produce a changed ciphertext, and submit it to the padding oracle. One byte of D2 might be obtained by testing every conceivable byte value for each round. The attack was built on top of the code that was supplied in the task. Through interaction with the padding oracle, the software ran through various byte values of CC1, testing the padding validity.

So, the plain text that we got in hexadecimal after running the code is:



So, if we convert the output from hexadecimal to a regular text, we get the encrypted message



Conclusion

In conclusion, the objective of this lab was to provide students with a hands-on experience of the padding oracle attack, a type of attack that exploits the behavior of systems that verify the validity of padding during decryption. This attack, first published by Serge Vaudenay in 2002, has been found to affect various well-known systems, including Ruby on Rails, ASP.NET, and OpenSSL.

The lab involved two oracle servers running inside a container, each containing a secret message encrypted with an unknown encryption key. The oracles provided the ciphertext and the Initialization Vector (IV) but did not disclose the plaintext or the encryption key. By sending modified ciphertexts to the oracle and analyzing the response, students were tasked with decrypting the secret message.

The lab covered important topics in cryptography, including secret-key encryption, encryption modes and paddings, and the padding oracle attack. The tasks included in the lab required students to analyze the paddings added to files of different lengths and to gradually decrypt the intermediate array D2, ultimately revealing the plaintext block P2.

The first task involved creating files of varying lengths and determining the paddings added to each file using the provided method. In the second task, we were allowed to stop after obtaining 6 bytes of D2, which would unlock the last 6 bytes of the plaintext block P2. Finally, the third task required us to automate the attack process and derive all the blocks of the secret message. The lab provided two padding oracle servers, one for each task level.

To exploit the vulnerability of the padding oracle, the attack involved modifying the first ciphertext block (CC1) and observing the oracle's response when decrypting the modified ciphertext alongside the second block (C2). By iteratively modifying CC1 and analyzing the response, the intermediate array D2 was gradually decrypted. The plaintext block P2 was then derived by XORing C1 and D2. While it was possible to write a program to derive all the blocks of the secret message in one run, we were also allowed to retrieve one block at a time.

Overall, the lab provided us with practical experience in understanding and exploiting the padding oracle vulnerability, enhancing our understanding of cryptographic systems and their security implications.

We found it really interesting to try to attack a server and find the hidden message, what really attracted us was that we need to think of an algorithmic and reasonable way to really find the hidden message meaning that we can't just decide that we want to attack without thinking in a functional way. Other thing was actually really beneficial was getting familiar with the docker environment since its like "trending" in nowadays job market and lots of companies care if students have it as a skill, so we were thrilled to explore it and learn how to interact with its environment.

Appendix:

task1

```
# echo -n "12345" > P1
```

```
# openssl enc -aes-128-cbc -e -in P1 -out C1
```

enter aes-128-cbc encryption password:

Verifying - enter aes-128-cbc encryption password:

Verify failure

bad password read

140654258586944:error:2807106B:UI routines:UI_process:processing
error:../crypto/ui/ui_lib.c:545:while reading strings

```
# openssl enc -aes-128-cbc -d -nopad -in C1 -out P1_new
```

Can't open C1 for reading, No such file or directory

140645939307840:error:02001002:system library:fopen:No such file or
directory:../crypto/bio/bss_file.c:69:fopen('C1','rb')

140645939307840:error:2006D080:BIO routines:BIO_new_file:no such
file:../crypto/bio/bss_file.c:76:

```
# ^C
```

```
# openssl enc -aes-128-cbc -e -in P1 -out C1
```

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.

```
# openssl enc -aes-128-cbc -d -nopad -in C1 -out P1_new
```

enter aes-128-cbc decryption password:

*** WARNING : deprecated key derivation used.

Using -iter or -pbkdf2 would be better.

```
# xxd P1_new
```

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

```
# ^C
```

```
# echo -n "1234567890" > P2
```

```

# openssl enc -aes-128-cbc -e -in P2 -out C2
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
Verify failure
bad password read
139772101162304:error:2807106B:UI routines:UI_process:processing
error:../crypto/ui/ui_lib.c:545:while reading strings
# openssl enc -aes-128-cbc -e -in P2 -out C2
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.
# openssl enc -aes-128-cbc -d -nopad -in C2 -out P2_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
# xxd P2_new
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890.....
# echo -n "1234567890123456" > P3
# openssl enc -aes-128-cbc -e -in P3 -out C3
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.
# openssl enc -aes-128-cbc -d -nopad -in C3 -out P3_new
enter aes-128-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
# xxd P3_new
00000000: 3132 3334 3536 3738 3930 3132 3334 3536  1234567890123456

```

00000010: 1010 1010 1010 1010 1010 1010 1010 1010

#

Taskk2.py

```
#!/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):
```

```
        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):
```

```
        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[0:16]
```

```
    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[:16] = C1[:16]
#####
# In the experiment, we need to iteratively modify CC1
# We will send this CC1 to the oracle and see its response.
CC1 = bytearray(16)

K = 1
for i in range(256):
    CC1[16 - K] = i
    for j in range(1, K):
        CC1[16 - j] = D2[16 - j] ^ K

    status = oracle.decrypt(IV + CC1 + C2)
    if status == "Valid":
        print("Valid: i = 0x{:02x}".format(i))
        print("CC1: " + CC1.hex())
        # Update D2 based on the value of K
        D2[16 - K] = i ^ K

K = 2
for i in range(256):
    CC1[16 - K] = i
    for j in range(1, K):
        CC1[16 - j] = D2[16 - j] ^ K

    status = oracle.decrypt(IV + CC1 + C2)
    if status == "Valid":
        print("Valid: i = 0x{:02x}".format(i))
        print("CC1: " + CC1.hex())
        # Update D2 based on the value of K
        D2[16 - K] = i ^ K

K = 3
for i in range(256):
    CC1[16 - K] = i
    for j in range(1, K):
        CC1[16 - j] = D2[16 - j] ^ K

```

```

status = oracle.decrypt(IV + CC1 + C2)
if status == "Valid":
    print("Valid: i = 0x{:02x}".format(i))
    print("CC1: " + CC1.hex())
    # Update D2 based on the value of K
    D2[16 - K] = i ^ K
K = 4
for i in range(256):
    CC1[16 - K] = i
    for j in range(1, K):
        CC1[16 - j] = D2[16 - j] ^ K

status = oracle.decrypt(IV + CC1 + C2)
if status == "Valid":
    print("Valid: i = 0x{:02x}".format(i))
    print("CC1: " + CC1.hex())
    # Update D2 based on the value of K
    D2[16 - K] = i ^ K
K = 5
for i in range(256):
    CC1[16 - K] = i
    for j in range(1, K):
        CC1[16 - j] = D2[16 - j] ^ K

status = oracle.decrypt(IV + CC1 + C2)
if status == "Valid":
    print("Valid: i = 0x{:02x}".format(i))
    print("CC1: " + CC1.hex())
    # Update D2 based on the value of K
    D2[16 - K] = i ^ K
K = 6
for i in range(256):
    CC1[16 - K] = i
    for j in range(1, K):
        CC1[16 - j] = D2[16 - j] ^ K

status = oracle.decrypt(IV + CC1 + C2)
if status == "Valid":
    print("Valid: i = 0x{:02x}".format(i))
    print("CC1: " + CC1.hex())
    # Update D2 based on the value of K
    D2[16 - K] = i ^ K

# Once you get all the 16 bytes of D2, you can easily get P2

```

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

```
#or just use a for loop
# Iterating K from 1 to 6
for K in range(1, 7):
    for i in range(256):
        CC1[16 - K] = i
        for j in range(1, K):
            CC1[16 - j] = D2[16 - j] ^ K

        status = oracle.decrypt(IV + CC1 + C2)
        if status == "Valid":
            print("Valid: i = 0x{:02x}".format(i))
            print("CC1: " + CC1.hex())
            # Update D2 based on the value of K
            D2[16 - K] = i ^ K

# Once you get all the 16 bytes of D2, you can easily get P2
P2 = xor(C1, D2)
print("P2: " + P2.hex())
```

```
Task3-automatedtask2.py
#!/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):
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s.connect((host, port))

        # Receive the ciphertext from the oracle
        ciphertext = self.s.recv(4096).decode().strip()
        self.ctext = unhexlify(ciphertext)

    def decrypt(self, ctext: bytes):
        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[0:16]
    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[:16] = C1[:16]
#####
# In the experiment, we need to iteratively modify CC1
# We will send this CC1 to the oracle and see its response.
CC1 = bytearray(16)

padding_found = False # Flag to indicate if valid padding is found

# Iterating K from 1 to 6
for K in range(1, 7):
    for i in range(256):
        CC1[16 - K] = i
        for j in range(1, K):
            CC1[16 - j] = D2[16 - j] ^ K

        status = oracle.decrypt(IV + CC1 + C2)
        if status == "Valid":
            print("Valid: i = 0x{:02x}".format(i))
            print("CC1: " + CC1.hex())
            # Update D2 based on the value of K
            D2[16 - K] = i ^ K
            padding_found = True
            break

    if not padding_found:
        print(f"Failed to find valid padding for K = {K}. Exiting.")
        break

# Once you get all the 16 bytes of D2, you can easily get P2
P2 = xor(C1, D2)
print("P2: " + P2.hex())

```

Task3.py

```
#!/usr/bin/python3
```

```

import socket
from binascii import hexlify, unhexlify

# XOR two byte arrays
def xor_bytes(first, second):
    return bytearray(x ^ y for x, y in zip(first, second))

class PaddingOracle:
    def __init__(self, host, port):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((host, port))

        # Receive the ciphertext from the oracle
        ciphertext = self.socket.recv(4096).decode().strip()
        self.ctext = unhexlify(ciphertext)

    def decrypt(self, ctext_bytes):
        # Send the ciphertext to the oracle for decryption
        self._send(hexlify(ctext_bytes))
        return self._receive()

    def _receive(self):
        # Receive and return the response from the oracle
        response = self.socket.recv(4096).decode().strip()
        return response

    def _send(self, hex_str_bytes):
        # Send the hexadecimal string to the oracle for decryption
        self.socket.send(hex_str_bytes + b'\n')

    def __del__(self):
        self.socket.close()

if __name__ == "__main__":
    # Create an instance of the PaddingOracle class
    oracle = PaddingOracle('10.9.0.80', 6000)

    # Get the IV + Ciphertext from the oracle
    iv_and_ctext = bytearray(oracle.ctext)
    iv = iv_and_ctext[0:16]
    block1 = iv_and_ctext[16:32]
    block2 = iv_and_ctext[32:48]
    block3 = iv_and_ctext[48:64]

    print("Block1: " + block1.hex())

```

```

print("Block2: " + block2.hex())
print("Block3: " + block3.hex())

# For holding the decrypted data
decrypted_blocks = [bytearray(16) for _ in range(3)]

# For holding the intermediate data
intermediate_blocks = [bytearray(16) for _ in range(3)]

# The ciphertext blocks and the IV
blocks = [iv, block1, block2, block3]

# Store decrypted plaintext blocks
plaintext_blocks = []

# Decrypt each block
for block_index in range(1, 4):
    for k in range(1, 17):
        found = False
        for i in range(256):
            intermediate_blocks[block_index-1][16 - k] = i
            for j in range(1, k):
                intermediate_blocks[block_index-1][16 - j] =
decrypted_blocks[block_index-1][16 - j] ^ k

            # Send the modified ciphertext to the oracle for
decryption
            status = oracle.decrypt(blocks[block_index-1] +
intermediate_blocks[block_index-1] + blocks[block_index])
            if status == "Valid":
                print(f"Valid for block={block_index} K={k}: i =
0x{i:02x}")
                print("Intermediate Block: " +
intermediate_blocks[block_index-1].hex())

                # Update decrypted_blocks based on the value of k
                decrypted_blocks[block_index-1][16 - k] = i ^ k
                found = True
                break

        if not found:
            print(f"Failed to find valid padding for
block={block_index} K={k}")
            break

```

```
    # Compute the plaintext block
    plaintext_block = xor_bytes(blocks[block_index-1],
decrypted_blocks[block_index-1])
    plaintext_blocks.append(plaintext_block)

# Print the decrypted plaintext blocks together
for idx, block in enumerate(plaintext_blocks):
    print(f"Plaintext Block {idx + 1}: {block.hex()}")
```