<u>Week 5 Homework due on Friday Oct 26, 18:59 Hour</u>

<u>Group 5</u>

Wong Ann Yi (1004000)
Liu Bowen (1004028)
Tan Chin Leong Leonard (1004041)

<u>Exercise 1</u>

Encrypt the following plaintext P (represented with 8-bit ASCII) using AES-ECB, with the key of 128-bit 0. You may use an existing crypto library for this exercise.

P = SUTD-MSSD-51.505*Foundations-CS*

a) What is the ciphertext C?
b) Swap the two blocks of C, what is the plaintext P1?
c) Change the last bit of C, what is the plaintext P2?
d) Discuss possible attacks against AES-ECB based on the results of a) - c).
e) How would you address those attacks?

Answers:

a) & b)

The codes are written in Python 3 and is as follows:

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
from Crypto.Cipher import AES
import codecs

decode_hex = codecs.getdecoder("hex_codec")
key = decode_hex('00000000000000000000000000000000')[0]
cipher = AES.new(key, AES.MODE_ECB)
ctext = cipher.encrypt('SUTD-MSSD-51.505*Foundations-CS*') # cipher text #
ctext = codecs.encode(ctext, 'hex')
print("The cipher text is:", ctext) # view the cipher text #
ctext = decode_hex(ctext)[0]
plain = cipher.decrypt(ctext) #convert back to plain text #
print("The plain text is:", plain) # view the plain text #
print("The modulo value of the text is:", len(ctext)%2) # check that the length of cipher text
is even #
print("The total length of the cipher text is:", len(ctext)/2) # find the total length of the cipher
texts #
B = int(len(ctext)/2)
C1 = ctext[:B] # first half of the cipher text #
```

```
C2 = ctext[B:] # second half of the cipher text #
C3 = C2+C1 # swap the first and second blocks of the cipher text #
C3 = codecs.encode(C3, 'hex')
print("The new cipher text is:", C3)
C3 = decode_hex(C3)[0]
plainnew = cipher.decrypt(C3) # decrypt the new cipher text #
print("The new plain text is:", plainnew) # view the new plain text #
ctext2  =  '885c4ce846078dea93b799e0bab3e710c97b2e6400a34bbde36f48684376dda9'  #
change the last bit of the original cipher text#
ctext2 = decode_hex(ctext2)[0]
ptext2 = cipher.decrypt(ctext2) # decrypt the changed cipher text #
print("The plain text after changing last bit of cipher text is:", ptext2)
```

^^

The outputs are:

```
The cipher text is:
b'885c4ce846078dea93b799e0bab3e710c97b2e6400a34bbde36f48684376dda8'
The plain text is: b'SUTD-MSSD-51.505*Foundations-CS*'
The modulo value of the text is: 0
The total length of the cipher text is: 16.0
The new cipher text is:
b'c97b2e6400a34bbde36f48684376dda8885c4ce846078dea93b799e0bab3e710'
The new plain text is: b'*Foundations-CS*SUTD-MSSD-51.505'
The plain text after changing last bit of cipher text is: b'SUTD-MSSD-
51.505\xcb/\x92k\x86\xc0T\xa0\\O\xf7\x8cn\xf8\xd8\xd6'
```

a) The cipher text C is:
885c4ce846078dea93b799e0bab3e710c97b2e6400a34bbde36f48684376dda8

b) The plain text P1 is: *Foundations-CS*SUTD-MSSD-51.505

c) After changing the last bit of cipher text, the new plain text P2 is:

"SUTD-MSSD-51.505\xcb/\x92k\x86\xc0T\xa0\\O\xf7\x8cn\xf8\xd8\xd6"

d) We call the phenomenon in c) as avalanche effect, which means any tiny change in input value can result in the irreversible change in output. Secondly, the AES ECB mode is encrypted each block one-by-one and therefore each ciphertext block is corresponding to its plaintext block. It cannot hide the plaintext mode. For example, repetitions in message may be shown in the ciphertext. Hence, attacker can make use of this disadvantage and can initiate an attack on the plaintext.

Lastly, as the plaintext is divided into each independent block (128 bits) and then encrypted. Attacker can attack some bits of plaintext which is harmful to users.

e) To address the above-mentioned attack, we can leverage the merits of CBC mode, encrypting each new block with former encrypted block. It's not so easy for attacker to initiatively attack plaintext compared with ECB mode. Also, we can use AES CFB mode which hides the plaintext mode and is difficult for an attacker to attack.

Its not so much about attacking. The value of n should be instead between 1 <= n <= b because if the plaintext i
a multiple of the block size b, then a whole new padding block of size b is added. This is necessary so the
deciphering algorithm can determine with certainty whether the last byte of the last block is a pad byte indicating
the number of padding bytes added or part of the plaintext message.

<u>Exercise 2</u>

The ciphertext (in hex)

87 F3 48 FF 79 B8 11 AF 38 57 D6 71 8E 5F 0F 91

7C 3D 26 F7 73 77 63 5A 5E 43 E9 B5 CC 5D 05 92

6E 26 FF C5 22 0D C7 D4 05 F1 70 86 70 E6 E0 17

was generated with the 256-bit AES key (also hex)

80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01

using CBC mode with a random IV. The IV is included at the beginning of the ciphertext. Decrypt this ciphertext. You may use an existing crypto library for this exercise.

Answers:

Using Python 3, we developed the below codes:

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
from Crypto.Cipher import AES
import codecs
decode_hex = codecs.getdecoder("hex_codec")
key =
decode_hex('8000000000000000000000000000000000000000000000000000000000000001')[
0]
IV = decode_hex('87F348FF79B811AF3857D6718E5F0F91')[0]
ciphertext =
decode_hex('7C3D26F77377635A5E43E9B5CC5D05926E26FFC5220DC7D405F1708670E6E017')
[0]
cipher = AES.new(key, AES.MODE_CBC, IV)
Plain = cipher.decrypt(ciphertext)
print(Plain)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

The output which is the plain text is:

```
b'Another secret!  And another.   '
```

Therefore, using the decrypted cipher text is: "Another secret! And another.   "

Exercise 3

With your AES-CBC implementation encrypt 160MB of zeros:

"\x00"*int(1.6*10**8)

under 128-bit long zeroed key and IV. What is the last 128 bits of the ciphertext? Compare efficiency (time) of your implementation with a chosen library or tool that offers AES-CBC.

Answers:

We will use Python 2.7 to write the below codes and solve the problem.

We use the Python Crypto library using the below:

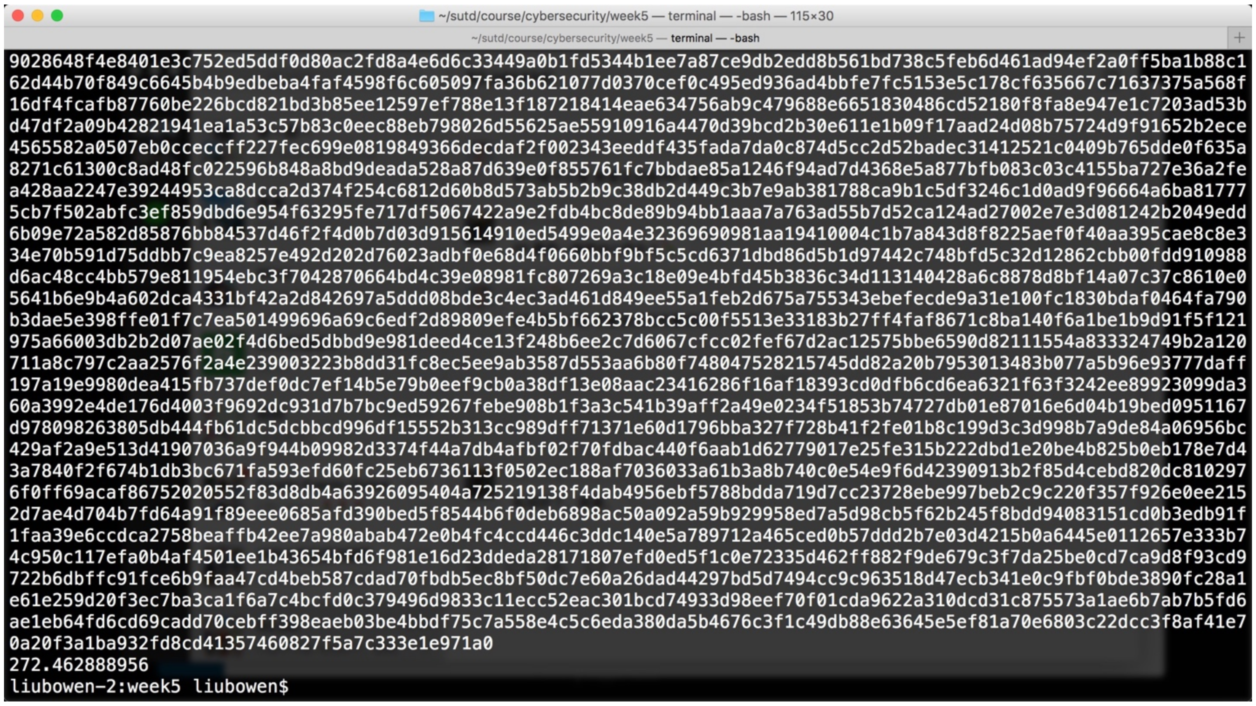
^^^

```
from Crypto.Cipher import AES
import binascii
import time
key = binascii.hexlify('\x00'*int(8))
IV = binascii.hexlify('\x00'*int(8))
def CBCLibrary():
   plaintex = ''
   for i in range (int(10**3)):
      for j in range(int(2*10**4)):
         plaintex += (binascii.hexlify('\x00')*int(8))
   cipher = AES.new(key, AES.MODE_CBC, IV)
return (cipher.encrypt(plaintex).encode('hex'))

start = time.time()
print CBCLibrary()
end = time.time()
print (end - start)
```

^^

We use hexlify() to generate 128-bit key and IV(as each hexlify() can output 00). CBC mode encrypt each 128-bit block therefore append each the number of 16 zeros(each hexlify() can output 00 and then multiply 8) and then recursive $2*10^4 * 10^3 = 2 * 10^7$ times(as $1.6*10^8/8=2*10^7$).
The last 128-bit ciphertext is: fd8cd41357460827f5a7c333e1e971a0

Figure 1 shows the full ciphertext.



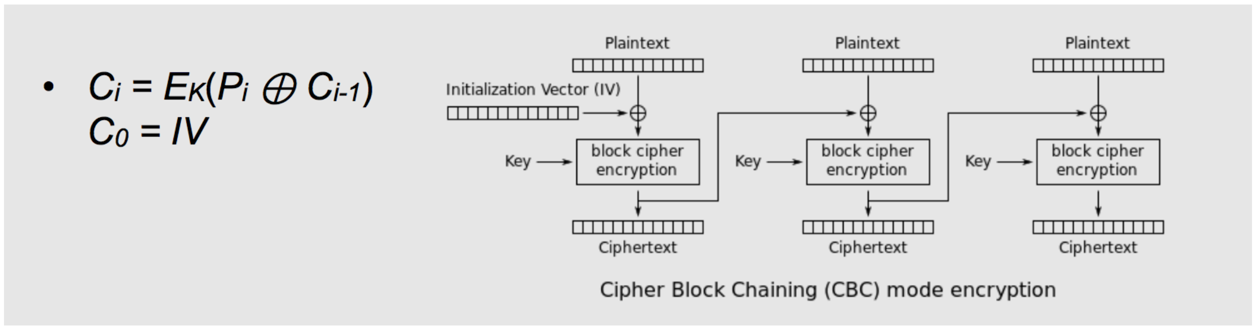
```
9028648f4e8401e3c752ed5ddf0d80ac2fd8a4e6d6c33449a0b1fd5344b1ee7a87ce9db2edd8b561bd738c5feb6d461ad94ef2a0ff5ba1b88c1
62d44b70f849c6645b4b9edbeba4faf4598f6c605097fa36b621077d0370cef0c495ed936ad4bbfe7fc5153e5c178cf635667c71637375a568f
16df4fcafb87760be226bcd821bd3b85ee12597ef788e13f187218414eae634756ab9c479688e6651830486cd52180f8fa8e947e1c7203ad53b
d47df2a09b42821941ea1a53c57b83c0eec88eb798026d55625ae55910916a4470d39bcd2b30e611e1b09f17aad24d08b75724d9f91652b2ece
4565582a0507eb0cceccff227fec699e0819849366decdaf2f002343eeddf435fada7da0c874d5cc2d52badec31412521c0409b765dde0f635a
8271c61300c8ad48fc022596b848a8bd9deada528a87d639e0f855761fc7bbdae85a1246f94ad7d4368e5a877bfb083c03c4155ba727e36a2fe
a428aa2247e39244953ca8dcca2d374f254c6812d60b8d573ab5b2b9c38db2d449c3b7e9ab381788ca9b1c5df3246c1d0ad9f96664a6ba81777
5cb7f502abfc3ef859dbd6e954f63295fe717df5067422a9e2fdb4bc8de89b94bb1aaa7a763ad55b7d52ca124ad27002e7e3d081242b2049edd
6b09e72a582d85876bb84537d46f2f4d0b7d03d915614910ed5499e0a4e32369690981aa19410004c1b7a843d8f8225aef0f40aa395cae8c8e3
34e70b591d75ddbb7c9ea8257e492d202d76023adbf0e68d4f0660bbf9bf5c5cd6371dbd86d5b1d97442c748bfd5c32d12862cbb00fdd910988
d6ac48cc4bb579e811954ebc3f7042870664bd4c39e08981fc807269a3c18e09e4bfd45b3836c34d113140428a6c8878d8bf14a07c37c8610e0
5641b6e9b4a602dca4331bf42a2d842697a5ddd08bde3c4ec3ad461d849ee55a1feb2d675a755343ebefecde9a31e100fc1830bdaf0464fa790
b3dae5e398ffe01f7c7ea501499696a69c6edf2d89809efe4b5bf662378bcc5c00f5513e33183b27ff4faf8671c8ba140f6a1be1b9d91f5f121
975a66003db2b2d07ae02f4d6bed5dbbd9e981deed4ce13f248b6ee2c7d6067cfcc02fef67d2ac12575bbe6590d82111554a833324749b2a120
711a8c797c2aa2576f2a4e239003223b8dd31fc8ec5ee9ab3587d553aa6b80f748047528215745dd82a20b7953013483b077a5b96e93777daff
197a19e9980dea415fb737def0dc7ef14b5e79b0eef9cb0a38df13e08aac23416286f16af18393cd0dfb6cd6ea6321f63f3242ee89923099da3
60a3992e4de176d4003f9692dc931d7b7bc9ed59267febe908b1f3a3c541b39aff2a49e0234f51853b74727db01e87016e6d04b19bed0951167
d978098263805db444fb61dc5dcbbcd996df15552b313cc989dff71371e60d1796bba327f728b41f2fe01b8c199d3c3d998b7a9de84a06956bc
429af2a9e513d41907036a9f944b09982d3374f44a7db4afbf02f70fdbac440f6aab1d62779017e25fe315b222dbd1e20be4b825b0eb178e7d4
3a7840f2f674b1db3bc671fa593efd60fc25eb6736113f0502ec188af7036033a61b3a8b740c0e54e9f6d42390913b2f85d4cebd820dc810297
6f0ff69acaf86752020552f83d8db4a63926095404a725219138f4dab4956ebf5788bdda719d7cc23728ebe997beb2c9c220f357f926e0ee215
2d7ae4d704b7fd64a91f89eee0685afd390bed5f8544b6f0deb6898ac50a092a59b929958ed7a5d98cb5f62b245f8bdd94083151cd0b3edb91f
1faa39e6ccdca2758beaffb42ee7a980abab472e0b4fc4ccd446c3ddc140e5a789712a465ced0b57ddd2b7e03d4215b0a6445e0112657e333b7
4c950c117efa0b4af4501ee1b43654bfd6f981e16d23ddeda28171807efd0ed5f1c0e72335d462ff882f9de679c3f7da25be0cd7ca9d8f93cd9
722b6dbffc91fce6b9faa47cd4beb587cdad70fbdb5ec8bf50dc7e60a26dad44297bd5d7494cc9c963518d47ecb341e0c9fbf0bde3890fc28a1
e61e259d20f3ec7ba3ca1f6a7c4bcfd0c379496d9833c11ecc52eac301bcd74933d98eef70f01cda9622a310dcd31c875573a1ae6b7ab7b5fd6
ae1eb64fd6cd69cadd70cebff398eaeb03be4bbdf75c7a558e4c5c6eda380da5b4676c3f1c49db88e63645e5ef81a70e6803c22dcc3f8af41e7
0a20f3a1ba932fd8cd41357460827f5a7c333e1e971a0
272.462888956
liubowen-2:week5 liubowen$
```

Figure 1

As for time cost, we use 'time' package and calculate the runtime before and after CBCLibrary().
Our test environment is: MacOS10.13 8GB memory, 128G SSD
the time for library is: 272.4628 seconds

For the pure CBC bottoms-up, we utilize the below concept and codes.

When writing pure CBC code, we need to generate P XOR C each time shown as below.



The code in Python 2.7 is shown as below:

^^^

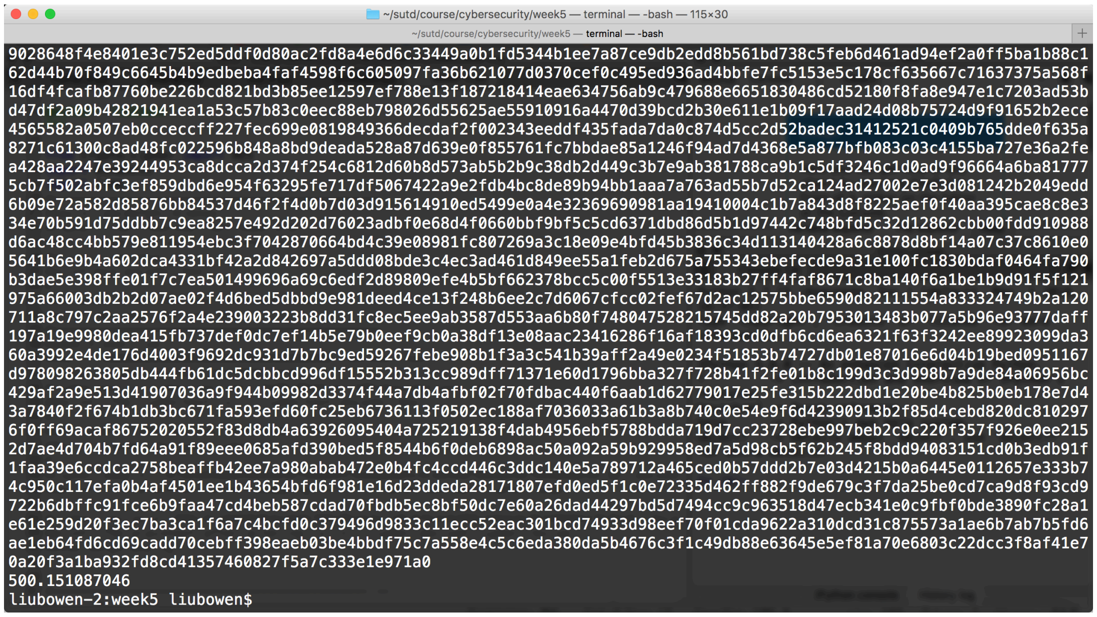
```
from Crypto.Cipher import AES
import binascii
import time
def CBCPure():

   res = ''
   C = ''
   for i in range(int(2*10**4)):
      for j in range(int(10**3)):
         cipher = AES.new(key, AES.MODE_CBC, IV)
         if i +j == 0:
            C0 = binascii.hexlify('\x00'*int(8))
            C = C0;
            P = binascii.hexlify('\x00'*int(8))
            res += cipher.encrypt(P).encode('hex')
            C = res;
            continue;
         P = binascii.hexlify('\x00'*int(8))
         size = len(format(int(P, 16) ^ int(C, 16), 'x'))
         if size == 32:
            xor_res = format(int(P, 16) ^ int(C, 16), 'x').decode('hex')
         else:
            diff = 32 - size;
            xor_res = ('0'*diff+format(int(P, 16) ^ int(C, 16), 'x')).decode('hex')
         C = cipher.encrypt(xor_res).encode('hex')
         res += C
   return (res)

start = time.time()
print CBCPure()
end = time.time()
print (end - start)
```

^^

The last 128-bit ciphertext is (similar to the previous output):
fd8cd41357460827f5a7c333e1e971a0

For the above test environment, the time taken: 500.1510 seconds. The full cipher text is in Figure 2.

Therefore, in summary the time taken to process using the Python Crypto library is 272.4628 seconds. In comparison, the time to process using the pure CBC bottoms up programming is 500.1510 seconds. Therefore, it is more efficient to run the encryption using a library tool.


~/sutd/course/cybersecurity/week5 — terminal — -bash — 115×30
~/sutd/course/cybersecurity/week5 — terminal — -bash

```
9028648f4e8401e3c752ed5ddf0d80ac2fd8a4e6d6c33449a0b1fd5344b1ee7a87ce9db2edd8b561bd738c5feb6d461ad94ef2a0ff5ba1b88c1
62d44b70f849c6645b4b9edbeba4faf4598f6c605097fa36b621077d0370cef0c495ed936ad4bbfe7fc5153e5c178cf635667c71637375a568f
16df4fcafb87760be226bcd821bd3b85ee12597ef788e13f187218414eae634756ab9c479688e6651830486cd52180f8fa8e947e1c7203ad53b
d47df2a09b42821941ea1a53c57b83c0eec88eb798026d55625ae55910916a4470d39bcd2b30e611e1b09f17aad24d08b75724d9f91652b2ece
4565582a0507eb0cceccff227fec699e0819849366decdaf2f002343eeddf435fada7da0c874d5cc2d52badec31412521c0409b765dde0f635a
8271c61300c8ad48fc022596b848a8bd9deada528a87d639e0f855761fc7bbdae85a1246f94ad7d4368e5a877bfb083c03c4155ba727e36a2fe
a428aa2247e39244953ca8dcca2d374f254c6812d60b8d573ab5b2b9c38db2d449c3b7e9ab381788ca9b1c5df3246c1d0ad9f96664a6ba81777
5cb7f502abfc3ef859dbd6e954f63295fe717df5067422a9e2fdb4bc8de89b94bb1aaa7a763ad55b7d52ca124ad27002e7e3d081242b2049edd
6b09e72a582d85876bb84537d46f2f4d0b7d03d915614910ed5499e0a4e32369690981aa19410004c1b7a843d8f8225aef0f40aa395cae8c8e3
34e70b591d75ddbb7c9ea8257e492d202d76023adbf0e68d4f0660bbf9bf5c5cd6371dbd86d5b1d97442c748bfd5c32d12862cbb00fdd910988
d6ac48cc4bb579e811954ebc3f7042870664bd4c39e08981fc807269a3c18e09e4bfd45b3836c34d113140428a6c8878d8bf14a07c37c8610e0
5641b6e9b4a602dca4331bf42a2d842697a5ddd08bde3c4ec3ad461d849ee55a1feb2d675a755343ebefecde9a31e100fc1830bdaf0464fa790
b3dae5e398ffe01f7c7ea501499696a69c6edf2d89809efe4b5bf662378bcc5c00f5513e33183b27ff4faf8671c8ba140f6a1be1b9d91f5f121
975a66003db2b2d07ae02f4d6bed5dbbd9e981deed4ce13f248b6ee2c7d6067cfcc02fef67d2ac12575bbe6590d82111554a833324749b2a120
711a8c797c2aa2576f2a4e239003223b8dd31fc8ec5ee9ab3587d553aa6b80f748047528215745dd82a20b7953013483b077a5b96e93777daff
197a19e9980dea415fb737def0dc7ef14b5e79b0eef9cb0a38df13e08aac23416286f16af18393cd0dfb6cd6ea6321f63f3242ee89923099da3
60a3992e4de176d4003f9692dc931d7b7bc9ed59267febe908b1f3a3c541b39aff2a49e0234f51853b74727db01e87016e6d04b19bed0951167
d978098263805db444fb61dc5dcbbcd996df15552b313cc989dff71371e60d1796bba327f728b41f2fe01b8c199d3c3d998b7a9de84a06956bc
429af2a9e513d41907036a9f944b09982d3374f44a7db4afbf02f70fdbac440f6aab1d62779017e25fe315b222dbd1e20be4b825b0eb178e7d4
3a7840f2f674b1db3bc671fa593efd60fc25eb6736113f0502ec188af7036033a61b3a8b740c0e54e9f6d42390913b2f85d4cebd820dc810297
6f0ff69acaf86752020552f83d8db4a63926095404a725219138f4dab4956ebf5788bdda719d7cc23728ebe997beb2c9c220f357f926e0ee215
2d7ae4d704b7fd64a91f89eee0685afd390bed5f8544b6f0deb6898ac50a092a59b929958ed7a5d98cb5f62b245f8bdd94083151cd0b3edb91f
1faa39e6ccdca2758beaffb42ee7a980abab472e0b4fc4ccd446c3ddc140e5a789712a465ced0b57ddd2b7e03d4215b0a6445e0112657e333b7
4c950c117efa0b4af4501ee1b43654bfd6f981e16d23ddeda28171807efd0ed5f1c0e72335d462ff882f9de679c3f7da25be0cd7ca9d8f93cd9
722b6dbffc91fce6b9faa47cd4beb587cdad70fbdb5ec8bf50dc7e60a26dad44297bd5d7494cc9c963518d47ecb341e0c9fbf0bde3890fc28a1
e61e259d20f3ec7ba3ca1f6a7c4bcfd0c379496d9833c11ecc52eac301bcd74933d98eef70f01cda9622a310dcd31c875573a1ae6b7ab7b5fd6
ae1eb64fd6cd69cadd70cebff398eaeb03be4bbdf75c7a558e4c5c6eda380da5b4676c3f1c49db88e63645e5ef81a70e6803c22dcc3f8af41e7
0a20f3a1ba932fd8cd41357460827f5a7c333e1e971a0
500.151087046
liubowen-2:week5 liubowen$
```

Figure 2

Exercise 4

Let P be the plaintext, and l(P) be the length of P in bytes. Let b the block size of the block cipher in bytes. Explain why the following is not a good padding scheme. a) Determine the minimum number of padding bytes necessary in order to pad the plaintext to a block boundary. b) This is a number n which satisfies $0 \leq n \leq b - 1$ and $n + l(P)$ is a multiple of b. c) Pad the plaintext by appending n bytes, each with value n.

Answers:

This padding scheme described in the question is also known as PKCS5.

If the block size of the data is 8 bytes and the message is only 4 bytes, then the padding required is 4 bytes:

8 bytes: FDFDFDFD --> FDFDFDFD04040404

The above padding method is not good because the block size required is padded with the bytes required. The attacker can easily guess the padding method used as it is not complicated and secure.

Another issue is if the plaintext *p* and *p* _ 0 have the same padded form it may even cause the padding to be irreversible. (Extracted from Cryptography Engineering: Design Principles and Practical Applications 1st Edition by Niels Ferguson (Author), Bruce Schneier (Author), Tadayoshi Kohno)

Padding can allow the attacker to gain knowledge of the traffic generated and plot for an attack even though the attacker may not know what is taking place in the background. (Extracted from https://en.wikipedia.org/wiki/Padding_(cryptography))

Padding messages make traffic analysis much harder as bits/bytes are added to the end of the messages. (Extracted from https://en.wikipedia.org/wiki/Padding_(cryptography))

Padding is that it makes ciphertext longer by at least one byte and at most a block. For example, if there are 15 plaintext bytes and a single byte missing to fill a block, padding adds a single 01 byte. If the plaintext is already a multiple of 16, the block length, add 16 bytes 10 (16 in decimal). (Extracted from Serious Cryptography (Jean-Philippe Aumasson)-Modes of Operation (Chapter 4))

Its not so much about attacking. The value of n should be instead between 1 <= n <= b because if the plaintext is a multiple of the block size b, then a whole new padding block of size b is added. This is necessary so the deciphering algorithm can determine with certainty whether the last byte of the last block is a pad byte indicating the number of padding bytes added or part of the plaintext message.

Exercise 5

Compare the security and performance advantages and disadvantages of each variant of CBC mode with a) fixed IV, b) counter IV, c) random IV, and d) nonce-generated IV.

Fixed IV Advantages:

Any part of the message can be encrypted and decrypted easily as it is using a fixed IV. (e.g. (e.g. Use IV = 10 for all the messages). The performance is fast as there is less overhead and processing required.

Fixed IV disadvantages:

It introduces the ECB problem for the first block of each message. If two different messages start with the same plaintext block, their encryptions will start with the same ciphertext blocks. In real life, messages often start with similar or identical blocks, and we do not want the attacker to be able to detect this. Extracted from Cryptography Engineering: Design Principles and Practical Applications 1st Edition by Niels Ferguson (Author), Bruce Schneier (Author), Tadayoshi Kohno (Page 66).

Counter IV Advantages:

Any part of the message can be encrypted and decrypted easily as it is using an incremental counter (e.g. Use IV = 0 for the first message, IV = 1 for the second message, etc).

Counter IV disadvantages

The attacker can promptly draw conclusions about the differences between the two messages, something a secure encryption scheme should not allow.

For example, the values 0 and 1 differ in exactly one bit. If the leading plaintext blocks of the first two messages also differ in only this bit (which happens much more often than you might expect), then the leading ciphertext blocks of the two messages will be identical. The attacker can promptly draw conclusions about the differences between the two messages, something a secure encryption scheme should not allow. Extracted from Cryptography Engineering: Design Principles and Practical Applications 1st Edition by Niels Ferguson (Author), Bruce Schneier (Author), Tadayoshi Kohno (Page 66).

Random IV advantages

It is more secure than Fixed IV and Counter IV as every IV generated will be at random, giving the attacker more problems trying to determine the IV value. A randomly chosen IV guarantees that even if the same message is sent repeatedly, the ciphertext will be completely different each time.

Finally, a randomly chosen IV prevents attackers from supplying chosen plaintext to the underlying encryption algorithm even if they can supply chosen plaintext to the CBC. This is extracted from Network Security: Private Communication in a Public World, Second Edition - Mike Speciner; Radia Perlman; Charlie Kaufma (chapter 4.2.2.).

<u>Random IV disadvantages</u>

The disadvantage of a random IV is that the ciphertext is one block longer than the plaintext. For short messages, this results in a significant message expansion, which is always undesirable.

For example

The recipient of the message needs to know the IV. The standard solution is to choose a random IV and to send it as a first block before the rest of the encrypted message. The resulting encryption procedure is as follows:

C0 := random block value Ci := E(K, Pi $\oplus$ Ci−1)
for i= 1, ... , k

with the understanding that the (padded) plaintext P1, ..., Pk is encrypted as C0, ..., Ck. Note that the ciphertext starts at C0 and not C1; the ciphertext is one block longer than the plaintext. The corresponding decryption procedure is easy to derive:

Pi := D(K, Ci) $\oplus$ Ci−1
for i = 1, ... , k

Extracted from Cryptography Engineering: Design Principles and Practical Applications 1st Edition by Niels Ferguson (Author), Bruce Schneier (Author), Tadayoshi Kohno (Page 67)

<u>Nonce-generated IV advantages</u>

It is secure just like the random IV. A nonce is a unique number which can be used once. The key necessary for message encryption is generated by encrypting the nonce. The performance is faster than Random IV as the extra information that needs to be included in the message is usually much smaller than in the random IV case. For most systems, a message counter of 32–48 bits are sufficient, compared to a 128-bit random IV overhead for the random IV solution. Extracted from Cryptography Engineering: Design Principles and Practical Applications 1st Edition by Niels Ferguson (Author), Bruce Schneier (Author), Tadayoshi Kohno (Page 68).

<u>Nonce-generated IV disadvantages</u>

Every number generated need to be unpredictable and not used before. A generator is required to keep generating numbers and track that it has not been used.

Exercise 6

An adversary observes the communication encrypted using CTR mode with the same fixed nonce. The nonce is hardcoded, so it is not included in the ciphertext. The adversary knows the following 16-byte ciphertext C

46 64 DC 06 97 BB FE 69 33 07 15 07 9B A6 C2 3D,

the following 16-byte ciphertext C'

51 7E CC 05 C3 BD EA 3B 33 57 0E 1B D8 97 D5 30,

and the plaintext P corresponding to C

43 72 79 70 74 6F 67 72 61 70 68 79 20 43 72 79.

What information, if any, can the adversary infer about the plaintext P' (corresponding to C')?

Answers:

We can derive the plaintext P′ corresponding to C′ by using the following relationship:

$P' = C \oplus P \oplus C'$

This is because the key is the same as the nonce is hardcoded and remained unchanged. Therefore, the key can be found by using the Xor operation on C and P. With the known key, we can find P' from C'.

The codes in Python 3 are:

^^

```
p = "43727970746F67726170687920437279"

c = "4664DC0697BBFE69330715079BA6C23D"

Cprime = "517ECC05C3BDEA3B33570E1BD897D530"

print("The Plain Text is:", bytes.fromhex(p))

p = int("43727970746F67726170687920437279", 16)

c = int("4664DC0697BBFE69330715079BA6C23D", 16)

Cprime = int("517ECC05C3BDEA3B33570E1BD897D530", 16)

Pprime = format((c^p^Cprime), 'x')

print("The Plain Prime Text in Hex is:", Pprime)

print("The Plain Prime Text is:", bytes.fromhex(Pprime))
```

^^^

The outputs are:

```
The Plain Text is: b'Cryptography Cry'

The Plain Prime Text in Hex is: 546869732069732061207365637265 74

The Plain Prime Text is: b'This is a secret'
```

The adversary uses the above method and codes and finds that the plain text P’ to be “This is a secret”.