The Hong Kong Polytechnic University

COMP 3334 – Computer Systems Security (Semester 2, Winter 2024)

Lab 1 – Cryptography

In this lab you will practice various basic data manipulation operations such as XORing data. You will also implement an attack on a XOR-based cipher to decrypt a given ciphertext automatically. You will then play with AES in CBC mode and attack the CBC mode of operation to bypass an authentication mechanism, which will require more careful thoughts about how this mode of encryption truly works.

Preliminaries

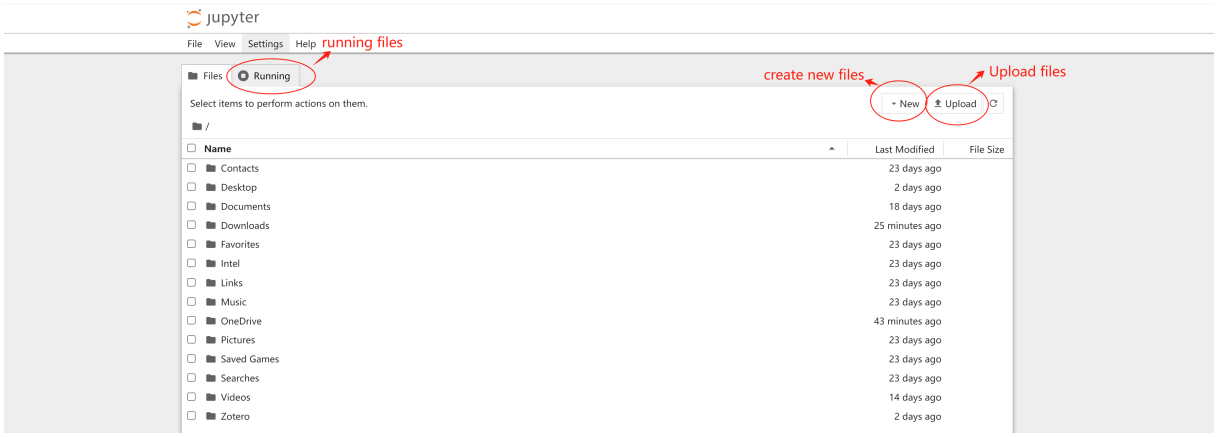
First you should know how to run Jupyter on the lab machines. Here are the steps of launching Jupyter.

Step 1: Enter start command:

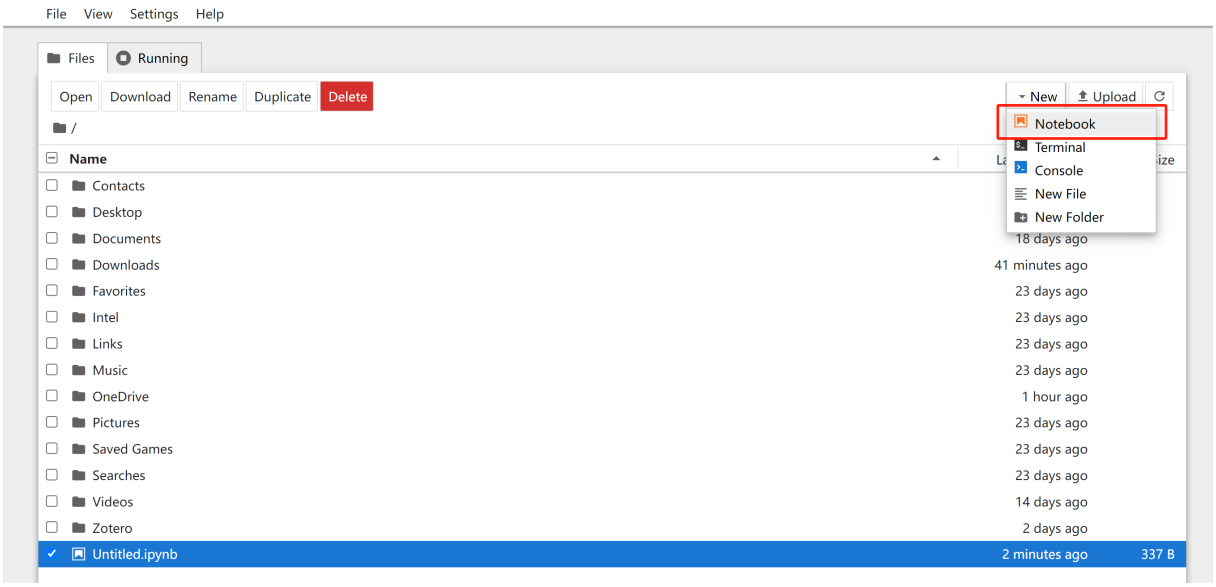
```
jupyter-notebook
```

```
^C
C:\Users\xinqi>jupyter-notebook
[I 2024-02-05 13:09:32.722 ServerApp] Extension package jupyter_server_terminals took 0.5760s to import
[W 2024-02-05 13:09:32.743 ServerApp] A `_jupyter_server_extension_points` function was not found in notebook_shim. Inst
ead, a `_jupyter_server_extension_paths` function was found and will be used for now. This function name will be depreca
ted in future releases of Jupyter Server.
[I 2024-02-05 13:09:32.744 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2024-02-05 13:09:32.746 ServerApp] jupyter_server_terminals | extension was successfully linked.
[I 2024-02-05 13:09:32.749 ServerApp] jupyterlab | extension was successfully linked.
[I 2024-02-05 13:09:32.751 ServerApp] notebook | extension was successfully linked.
[I 2024-02-05 13:09:32.755 ServerApp] Writing Jupyter server cookie secret to C:\Users\xinqi\AppData\Roaming\jupyter\run
time\jupyter_cookie_secret
```

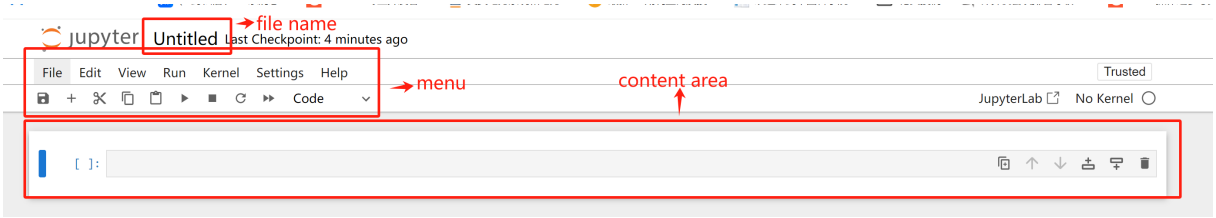
Step 2: After successful startup, a window opens in your browser. You can check running files, upload files, create new files, etc.



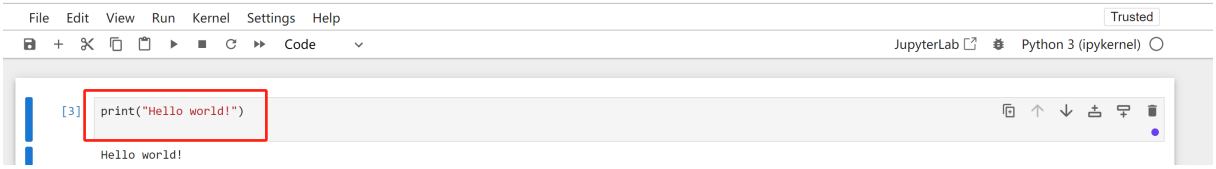
Step 3: Create notebook. When you click "New", you can create a notebook.



Step 4: After creating the notebook, you can go to the notebook page. It consists of three main areas: file name, menu bar, and content editing.



Step 5: Running python code.



Exercises

To write the code needed in the following exercises, you will need to search online in the Python documentation (we suggest appropriate packages as well). Say, if you don't know how to make a for loop in Python given a specific range, you need to search. Similarly, check online if you don't know how to utilize sets/dictionaries {}, lists `()`, and sequences `[]`. The data structures tutorial is a good start.

Exercise 1: Base64 [20min]

Objective. The objective of this experiment is to understand the base64 encoding, and to explore its applications in computer science and data transmission.

What is base64? Base64 is a way in which 8-bit data is encoded into a format that can be represented in 6 bits. This is done using only the characters A-Z, a-z, 0-9, +, and / in order to represent data, with = used to pad data. The base64 encoded data ends up being longer than the original data: for every 3 bytes of data, it produces 4 bytes of encoded data. This is due to the fact that we are squeezing the data into a smaller set of characters. The term base64 is taken from the Multipurpose Internet Mail Extensions (MIME) standard, which is widely used for HTTP and XML, and was originally developed for encoding email attachments for transmission.

Encoding data into base64 is carried out in multiple steps, as follows:

Step 1: The base64 encoding algorithm receives an input stream of 8-bit bytes. For a plain English text, each letter maps to one or more bytes depending on the character encoding. By following the simple ASCII encoding, each letter corresponds to a single byte. For instance, the letter 'o' is encoded in ASCII as 111, or 01101111 in binary. See the ASCII table here.

```
#input:
-> oz9un
#Binary Representation (8-bit sequences):
-> 01101111 01111010 00111001 01110101 01101110
```

Step 2: The 8-bit groups are split into 6-bit groups, and the last group is padded '0's at the end as necessary to obtain a full 6-bit group.

```
#Binary Representation (6-bit sequences):
-> 011011 110111 101000 111001 011101 010110 111000 #Pad with two '0' at the end
```

Step 3: Each 6-bit group represents an index between 0 and 2^6-1 (=63), which then maps to a character of the base64 alphabet, shown in Fig. 1.

```
#The 6-bit groups above correspond to:
-> 27 55 40 57 29 22 56
```

Step 4: Map these numbers to their corresponding base64 character according to Fig. 1:

```
#The indexes then map to:
-> b3o5dW4
#The result is padded with '=' to make it a multiple of 4 characters:
-> b3o5dW4=
```

Decoding is performed in reverse. After removing the '=' pad, each base64 character is mapped back to a 6-bit index. All indexes are reorganized as 8-bit bytes, discarding the last incomplete byte. A string is eventually decoded from the sequence of bytes.

Your task. In this Exercise, you should implement string encoding and decoding using base64 by leveraging the `base64` package. Python's string type uses the Unicode Standard for representing characters. To operate on bytes, you first need to encode the string into a bytes representation of the Unicode string. This is done using the `encode()` function on a string, typically with the UTF-8 character encoding, which generalizes ASCII to many more symbols.

Look into the documentation of the packages and functions given above.

Base64 alphabet defined in RFC 4648.

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Figure 1: Base64 alphabet defined in RFC 4648

Expected outcome. You can use the following example to test the correctness of your code.

```
#input string:
-> sdifusdf9a8sdfsadfas
#base-64 encoded string:
-> c2RpZnVzZGY5YThzZGZzYWRmYXM=
#decoded string:
-> sdifusdf9a8sdfsadfas
```

Exercise 2: Fixed XOR [20min]

Objective. Your task is to implement a function that can take two equal-length buffers (i.e. hexadecimal strings) and produce their XOR combination.

Background. Before diving into the exercise, it is essential to understand the XOR (exclusive OR) operation, which is a fundamental component in many cryptographic algorithms. XOR operates on bits, with the following properties:

Commutative: $A \oplus B = B \oplus A$
Associative: $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
Identity: $A \oplus 0 = A$
Self-Inverse: $A \oplus A = 0$
Below is the XOR truth table:

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Your task. Your job is to implement the function `XOR` that fulfils the following requirements:

1. It must take two strings as input, each representing hexadecimal values.
2. Both strings will be of equal length.
3. The function should decode the strings from hexadecimal. Check out the `bytes.fromhex()` and related functions to decode hex-encoded strings and recode strings.
4. Then, it should compute a bit-wise XOR of the two binary strings. The bit-wise $\oplus$ operator is "^". Check out the `zip()` function to write things more efficiently.
5. Finally, the output should be encoded back into a hexadecimal string.

Expected outcome. To verify the correctness of your function, we provide a test case with expected output:

- Input string 1: `1c0111001f010100061a024b53535009181c`
- Input string 2: `686974207468652062756c6c277320657965`
- Expected output: `746865206b696420646f6e277420706c6179`

Now, try to display the decoded output string :)

Exercise 3: Single-byte XOR cipher [40min]

Objective. Your task is to decrypt an English message that has been encrypted using a single-byte XOR cipher. The message is given in hexadecimal form and has been XORed against a single character. You must find the key and decrypt the message.

Your task. To decrypt the message, you will need to:

1. Write a function that will XOR the given hex-encoded string against a single byte. See how you can modify Exercise 2 for this task.
2. Develop a scoring system to evaluate the resulting plaintext.

   Hint: What are the most frequently occurring characters in English?

   Hint 2: The message is fairly small, expected top frequencies may be different here.

   Hint 3: To count frequencies, see `collections.Counter`.
3. Iterate through all possible single-byte keys (0x00 to 0xFF), apply the XOR function, and score the outputs.
4. Output only the decrypted string with the highest score.

Expected outcome:

A function `crackXOR(ct)` that takes as argument a hex-encoded encrypted English message `ct`, and returns the pair (key *as integer*, decrypted_message *as string*). Your function should be able to decrypt the following message:

`1b37373331363f782a373b332b783931367f2c783f312e3d783f37373c783a2a372c3076`

Exercise 4: CBC bitflipping attack [40–60min]

Objective. The objective of this exercise is to implement CBC bitflipping attacks, which is helpful to understand this mode of operation.

Background. CBC (Cipher Block Chaining) is a mode of operation for block ciphers. It is considered cryptography strong; however, it comes with certain caveats. Padding is required when using this mode, and there is an parameter called Initialization Vector (IV) that is used in the encryption of the first block. The IV must be random. IVs do not need to be kept secret and they can be included in a transmitted message. They must also be the same length as the block size of the cipher. Each time something is encrypted, even with the same key, a new IV should be generated. You should never an IV with a given key, and particularly never use a constant IV.

There are certain advantages of the CBC mode. First, the ciphertext blocks are not only related to the current plaintext block but also to the previous ciphertext block (or IV for for first block), hiding the statistical characteristics of the plaintext before encryption. Second, it has a limited two-step error propagation property, meaning that a change in one bit of a ciphertext block will only affect the current ciphertext block and the next ciphertext block. Third, it has self-synchronization property, which means that if the ciphertext from block i onwards is correct, then the decryption of blocks $i + 1$ onwards will be successful. However, CBC exhibits an interesting behavior that could be misused by an attacker to manipulate parts of the decrypted ciphertext, which we are going to show in this exercise.

The encryption and decryption operations in CBC mode are shown in the following figures. During encryption, $C_i = \mathrm{E}_k(P_i \oplus C_{i-1})$. During decryption, $P_i = C_{i-1} \oplus \mathrm{D}_k(C_i)$.

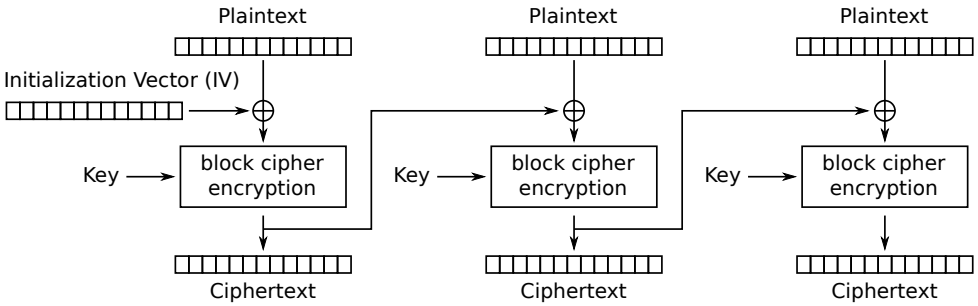


Figure 2: Cipher block chaining (CBC) mode of operation (during **encryption**)

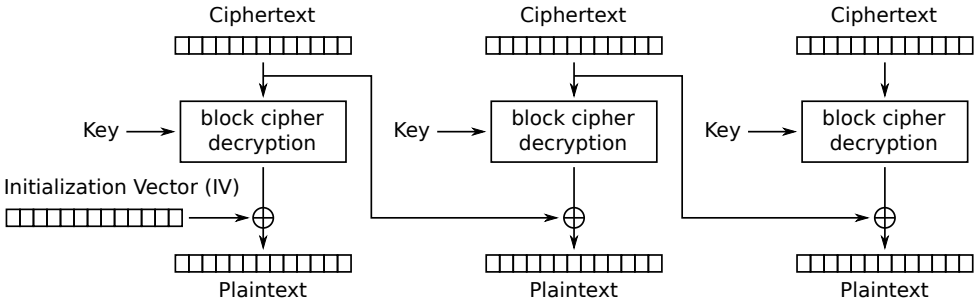


Figure 3: Cipher block chaining (CBC) mode of operation (during **decryption**)

Encrypt using AES-CBC. The following code implements the required functions to encrypt and decrypt any data using AES in CBC mode. When the length of the data is not a multiple of the block size, it is padded following the PKCS#7 padding standard. Padding simply means that we add data as necessary to perform encryption, but the data should be removed after decryption because it is not part of the plaintext. The AES cipher is provided in Python in the `cryptography` package. You can read more about PKCS#7 padding in the documentation.

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
BLOCKLEN = 16

# pad data using PKCS#7 to the nearest multiple of the block size
def pad(data, block_size):
    padder = padding.PKCS7(block_size * 8).padder()
    padded_data = padder.update(data) + padder.finalize()
    return padded_data

# unpad data
def unpad(padded_data, block_size):
    unpadder = padding.PKCS7(block_size * 8).unpadder()
    data = unpadder.update(padded_data) + unpadder.finalize()
    return data

# pad and encrypt plaintext with AES in CBC mode with given key and IV
def AESencrypt(plaintext, key, iv):
    # pick an instance of AES in CBC mode
    aes = Cipher(algorithms.AES(k), modes.CBC(iv)).encryptor()
    return aes.update(pad(plaintext, BLOCKLEN)) + aes.finalize()

# decrypt and unpad ciphertext
def AESdecrypt(ciphertext, key, iv):
    # pick an instance of AES in CBC mode
    aes = Cipher(algorithms.AES(k), modes.CBC(iv)).decryptor()
    return unpad(aes.update(ciphertext) + aes.finalize(), BLOCKLEN)

# pretty-print encrypted blocks of data
def blocks(data):
    split = [bytes.hex(data[i:i + BLOCKLEN]) for i in range(0, len(data), BLOCKLEN)]
    return ' '.join(split)
```

Try to use the `AESencrypt` and `AESdecrypt` functions using the following code:

```
from os import urandom

# pick a key
k = urandom(16)
print("k = %s" % bytes.hex(k))

# pick an IV
iv = urandom(16)
print("iv = %s" % bytes.hex(iv))

msg = b"a secret message"
ct = AESencrypt(msg, k, iv)
print("enc(%s) = %s" % (blocks(msg), blocks(ct)))
pt = AESdecrypt(ct, k, iv)
print("dec(%s) = %s" % (blocks(ct), blocks(pt)))
```

Your task. Assume a website keeps track of user information in a cookie encrypted by a secret key. The user's browser will record the cookie and attach it to every request. Your first task is to form the cookie as follows. Write a function that takes an arbitrary input string, prepend (add before) the string:

`comment1=cooking-rocks;userdata=`

then append (add at the end) the string:

`;comment2=%20ain%27t%20give%20good%20broth`

The function should quote out the ";" and "=" characters (to e.g., "";"" and ""="").

Your second task is to decrypt the string and look for the string ";admin=true;", which would grant the user admin privileges. However, such a string should be impossible to obtain in our setup since the `userdata` is sanitized using quotes around some of the required characters. You need to carefully flip bits in the plaintext to complete this task. The principle of this bitflipping attack is very simple, we can observe in more detail the decryption process to find the following characteristics:

- The IV only influences the first plaintext block.
- A block k of ciphertext, C_i, influences the block $i+1$ of plaintext, P_{i+1}.

Make sure you understand how the characteristics work.

Next, the goal of the bitflipping attack is to change C_i into $C_i \oplus P_{i+1} \oplus A$, so that decrypting the next ciphertext C_{i+1} will give you $\mathrm{D}_k(C_{i+1}) \oplus (C_i \oplus P_{i+1} \oplus A) = P_{i+1} \oplus P_{i+1} \oplus A = A$. Therefore, the decrypted ciphertext of block $i+1$ will be entirely under our control and will take the value A that we chose. The following example code can help in understanding the bitflipping theory of the work, you can test it in your Jupyter Notebook:

```
pt = prepare_cookie(b"blah") # the function need to be implement at your first task.
ct = AESencrypt(pt, key, iv)
# flip the least significant bit in the first byte of the ciphertext
modified_ct = bytes([ct[0] ^ 0x01]) + ct[1:]
# see the difference by XORing both ciphertexts
ct_xor_modifiedct = bytes([x ^ y for x, y in zip(ct, modified_ct)])
print("ct XOR modified_ct = %s" % blocks(ct_xor_modifiedct))
```

```
# decrypt the new ciphertext
new_pt = AESdecrypt(modified_ct, k, iv)
print("before: dec(%s) = %s" % (blocks(ct), blocks(pt)))
print("after : dec(%s) = %s" % (blocks(modified_ct), blocks(new_pt)))
# see the difference in the resulting plaintext
pt_xor_newpt = bytes([x ^ y for x, y in zip(pt, new_pt)])
print("pt XOR new_pt = %s" % blocks(pt_xor_newpt)
```

```
print("\"%s\" becomes \"%s\"" % (pt[16:32].decode(), new_pt[16:32].decode()))
```

Now is your turn! Devise a value for `userdata` that can give you an advantage, and figure out which block of ciphertext to modify and how.

To help you confirm your attack is successful, please use the following code to confirm it. Note that the `assert` function needs to pass without errors.

```
def decrypt_oracle(bs, k, iv):
    return is_admin(AESdecrypt(bs, k, iv))

def is_admin(bs):
    return b';admin=true;' in bs

assert decrypt_oracle(malicious_ct, k, iv) == True
```

Note:

Some hints for this exercise:

- Consider userdata = "AadminAtrueA" to design your attack.
- Change the ciphertext of each corresponding "A" byte in the block prior to the userdata's block.

Need help?

Teaching Assistants you can contact about this lab:

1. Xuyuan CAI, xuyuan.cai@connect.polyu.hk, exercise 1
2. Jinan Jiang, jinan.jiang@connect.polyu.hk, exercises 2 and 3
3. Bowen CUI, bowen.cui@connect.polyu.hk, exercise 4.