THE UNIVERSITY OF BRITISH COLUMBIA
Electrical and Computer Engineering Department

CPEN 442 **Introduction to Cybersecurity** **Winter Term 1, 2024**

Assignment 2: Stream and Block Ciphers

Due date: **Friday, September 27, 2024, 11:59 pm**
Hard deadline: **Sunday, September 29, 2024, 11:59 pm**

Please use Piazza for questions and clarifications. Instructor and TA help will be provided only until 2pm on the same day as the *due date* above. This assignment has a 48-hour automatic deadline extension to account for unexpected issues when uploading the assignment. You are allowed to submit the assignment until the *hard deadline* date above, with no penalization. No exceptions after this date will be made!! If you have a long-term issue that will prevent you, or your group, from submitting the assignment, let the instructor know *well before the due date.*

Total marks: 100 pts
TA: no TA `[simon.oya@ubc.ca]`

Doing the assignment: This assignment can be done either **individually** or in **groups of two**. Regardless of what you do, you must join a group set in Canvas (check the `People` section, then `A2 group`). The group can be just yourself.

What to submit:

- A PDF file called `a2.pdf`, with the answer to Questions 2.4 and 2.5 (everything else is coding).
- A Python file `a2.py`; please see the provided template and do not change the function headers.

1 Handling strings and bytes in Python

1.1 Strings and bytes

Before we start the crypto part of this assignment, it is good to have a solid understanding of strings and bytes in Python. You all know what a Python string is: a (usually readable) set of characters enclosed by single or double quotation marks, e.g.,

```
text='hello world!'
```

The process of converting Python strings into bytes is a type of *encoding*; converting bytes back to characters is *decoding*. There are many types of encoding formats (e.g., UTF-8, ASCII). In this course, we will use ASCII. Let's convert the text above into a bytes object:

```
text_bytes = text.encode('ascii')
```

The result is a `bytes` object. If you print this object, you will see the text again, preceded by a `b`. This is just Python interpreting the `bytes` object as a string. Also note that we can directly input the bytes as text if we declare the object as follows:

```
text_bytes=b'hello world!'
```

To convert the bytes back to text, we would use

```
text_string = text_bytes.decode('ascii')
```

1.2 Base64 encoding

Many times, we want to transfer bytes of data through a medium that is supposed to contain text data (e.g., a text file, like the ones we will use in this assignment). Base64 encoding is a technique to interpret the bytes as groups of 6 bits instead, and then we can represent each group of 6 bits as a character. It does not matter what these bytes originally represented (text, images, video, etc.). When we encode something into Base64 characters, we will (most likely) get something "unreadable". The Base64 characters are `[A-Z]`, `[a-z]`, `[0,9]`, `+` and `/` (let's count how many characters this is: $26 + 26 + 10 + 2 = 64$ characters, which is what we can represent with 6 bits).

To convert `bytes` to Base64 and back, we will use `base64` package. For example, let's take the bytes above (`text_bytes`) and encode them as Base64:

```python
import base64
text_bytes=b'hello world!'
text_base64 = base64.b64encode(text_bytes)
print(text_base64)
```

Note that `text_base64` is another `bytes` object, but it is easy to store as text (we can convert it to text by using `.decode('ascii')`, as above). This is not meant to be read by a human, it is just a way of storing binary data in 6-bit groups (6 bits per character). When the bytes object that you are trying to encode is not a multiple of 6 (bits), you will get some "padding" at the end (this is the equal sign `=` that you see in this case).

The decoding process is similar:

```python
text_binary = base64.b64decode(text_base64)
text = text_binary.decode('ascii')
```

Here, we decoded the Base64 string into bytes, and then decoded the bytes as the original string.

1.3 Checking that you got this working

To check that you got this working, check that you know how to read the text (ASCII) that is encoded in this Base64 string:

```python
text_base64 = 'WW91ciBkZWNvZGluZyBzZWVtcyB0byBiZSB3b3JraW5nIGNvcnJlY3RseSEh'
```

To summarize:

$$\begin{tikzcd}
\texttt{'hello!'} \arrow[r, "\texttt{ASCII encode}"] & \texttt{b'hello!'} \arrow[r, "\texttt{b64encode}"] & \texttt{b'aGVsbG8h'} \arrow[r, "\texttt{ASCII decode}"] & \texttt{'aGVsbG8h'}
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{'hello!'} & \texttt{b'hello!'} \arrow[l, "\texttt{ASCII decode}"'] & \texttt{b'aGVsbG8h'} \arrow[l, "\texttt{b64decode}"'] & \texttt{'aGVsbG8h'} \arrow[l, "\texttt{ASCII encode}"']
\end{tikzcd} \tag{1}$$

1.4 XORing bytes

In this assignment, you will need to XOR bytes many times. Let's implement this function.

[5 pts] **Question 1.1:** Inside `a2.py`, fill in the function `xor_bytes`, which takes two `bytes` objects and returns their XOR as another `bytes` object. If you don't know how to

do this, look on the Internet. This is usually one line of code. You can decide what the behavior is when the inputs are not of the same length, it is up to you (e.g., raise an error or truncate to the length of the shortest input; the latter will be more convenient later on).

To check that this is working, if you take the following Base64 strings and XOR them, you should get something readable:

```python
data1 = 'QHAl2CEK9djY+WCq'
data2 = 'CBlBvERk1ay9gRSL'
```

If it is not working: remember to decode the Base64 representation before XORing (so that we get regular bytes).

2 Implementing different modes of operation for AES

For this assignment, we will use the AES cryptosystem. There are many ways to do this. In this assignment, we will use PyCryptodome. Install this package first.

The code below shows how to use PyCryptodome to encrypt a plaintext (`plaintext` is a `bytes` object) using a randomly-generated key (that we generated with `randbytes`):

```python
from Crypto.Cipher import AES
from random import randbytes

plaintext = b'hello, world!!!!!'
key = randbytes(AES.block_size)
cipher = AES.new(key, AES.MODE_ECB)
ciphertext = cipher.encrypt(plaintext)
```

Make sure you understand what this code is doing. For example, print `AES.block_size` and think if this aligns with what we have seen in the classroom.

In particular, try to encrypt a shorter plaintext, e.g., `b'hello, world!'` and see what happens. Indeed, when the input bytes are not a multiple of the block size, we cannot encrypt... unless we pad the plaintext.

Padding is simply filling up the plaintext until its length in bytes is a multiple of the block size (in bytes). There are many ways to pad. One of the most popular and simplest padding mechanism is called PKCS#7. For brevity, this document will not explain how it works, but it is very simple and Wikipedia explains it very clearly.

To pad using PyCryptodome, we do:

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

plaintext = b'hello, world!'
# Pad
padded_plaintext = pad(plaintext, AES.block_size)
print(padded_plaintext)
# Encrypt
key = randbytes(AES.block_size)
cipher = AES.new(key, AES.MODE_ECB)
ciphertext = cipher.encrypt(padded_plaintext)
print(ciphertext)
```

As you can see, we needed to pad the plaintext with three bytes (`b'hello, world!'` is 13 characters – or bytes – long). Therefore, we padded with a binary 3 (00000011), which is represented as `'\x03'` when printed. You can also take a look at what the ciphertext looks like when printing the bytes directly. As a human, it is hard to see how many bytes this ciphertext is. When we represent it with Base64 characters it looks better, right?

2.1 AES in ECB mode

[5 pts] Question 2.1: Program the functions `encrypt_aes_ecb` and `decrypt_aes_ecb`, which encrypt and decrypt using AES in ECB mode. These functions must also pad and unpad. Notice that the plaintext, ciphertex, and key are all `bytes` objects. You basically have all the code you need above. I should not take more than 2-3 lines per function.

To verify that this works: decrypt this ciphertext with the given key, and you should get something readable.[1]

```python
key = 'zQcs2L5vn2KsTAnCggbn4w=='
ciphertext = '2466mVLtcnf1FuvQiryyVHA3v2mJ8pUEEI0X7kaQHUzVOj
    kApO6E1mSsleGPa7ywM7P4rUaKYISQsT4LnUbVVQ=='
```

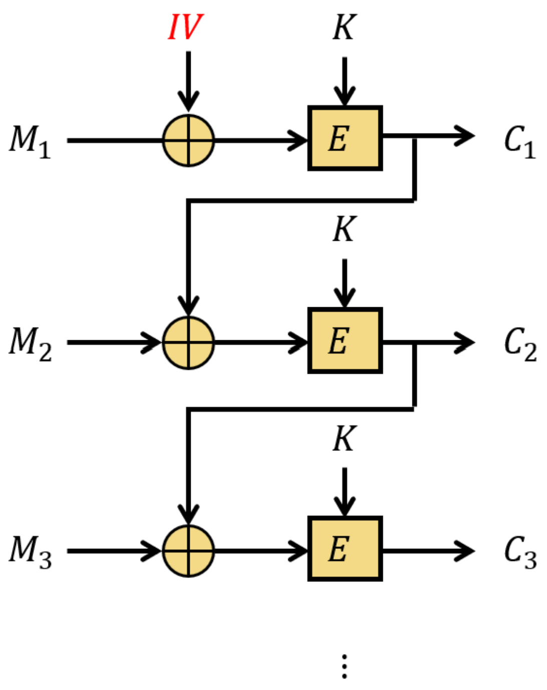
2.2 AES in CBC mode

[10 pts] Question 2.2: This is the first "real" question. You have to implement AES in CBC mode by filling the functions `encrypt_aes_cbc` and `decrypt_aes_cbc`. Do *not* use

[1]This is the last time the assignment will mention this: these strings are in Base64. You should convert them to bytes first. Also, that ciphertext is supposed to be a single line (I had to add a line break so that it would fit in the pdf).

PyCryptodome's implementation of CBC mode!! Instead, define the cipher in ECB mode, as we have been doing (`cipher = AES.new(key, AES.MODE_ECB)`), and use this `cipher` to implement CBC.

To do this, remember the structure of CBC:



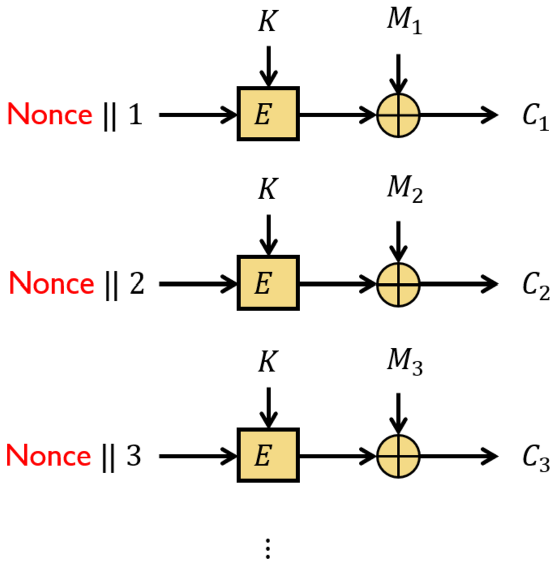
You will have to manipulate `bytes` objects. These work very similarly to Python strings: you can write `plaintext[i]` to take the ith byte of `plaintext`, and adding `bytes` objects together simply concatenates them. Remember to also pad the plaintext first, and use the IV for the first block.

Note: the function should work for valid inputs, and the behavior is unspecified for invalid inputs (e.g., the IV or key are the wrong size). You do not need to take corner cases into account.

To verify your script, you can compare that the behavior is the same as PyCryptodome's implementation (e.g., see an example on how to use Pycryptodome's AES in CBC mode in the documentation.)

2.3 AES in CTR mode

[10 pts] **Question 2.3:** Finally, implement AES in CTR mode by filling the functions `encrypt_aes_ctr` and `decrypt_aes_ctr`.[2] As before, define the cipher in ECB mode (`cipher = AES.new(key, AES.MODE_ECB)`) and put the pieces together to get CTR mode.



Some notes:

- Assume the IV (i.e., the nonce) is 64 bits long (8 bytes), and the counter is also 64 bites long (8 bytes), so that together they are 16 bytes, which is the block size for AES.
- To increase the counter, you cannot do it directly as a `bytes` object. You can look into how to convert it to `int` and back, for example.
- AES-CTR does not use padding. For the last "block", you might have generated more keystream bytes than the plaintext has. Just ignore the final bytes of the keystream. For example, in the image above, let's assume M_3 is 10-byte long. In that case, you will generate a 16-byte keystream by doing $E_K(\text{Nonce}\|3)$, but you should only use the first 10 bytes of this keystream to XOR with M_3.

Verify that the behavior is the same as PyCryptodome's implementation (notice that the IV is called nonce here, e.g., `cipher = AES.new(key, AES.MODE_CTR, nonce=iv)`)

[2] You'll realize that one of these functions will be very short :).

2.4 Detecting modes of operation

In the classroom, we saw that ECB has one particular property that makes it very weak when used to encrypt more than one block. In file `q24_plaintext.txt`, you have the lyrics of a song.[3] Files `q24_ciphertext1.txt`, `q24_ciphertext2.txt`, and `q24_ciphertext3.txt` contain an encryption of this plaintext with a random key: one is using AES-ECB, another one AES-CBC, and another one AES-CTR.

[5 pts] **Question 2.4:** Guess which one is AES-ECB. You do not need to crack the key to make this guess (remember, the sun will die before you do). In `a2.pdf`, explain which one is AES-ECB and how you guessed this.

[5 pts] **Question 2.5:** Guess which one is AES-CTR. Guessing which one is CTR is possible in this particular case. You do not need to code anything for this: just looking at the ciphertexts should be enough. In `a2.pdf`, explain which one is AES-CTR and how you guessed this.

3 An attack against AES-CTR

Now for the fun part. Let's start by a simple integrity attack against stream ciphers. Remember a property of stream ciphers: if we XOR anything with the ciphertext, this is equivalent to XORing with the plaintext (e.g., slide 30 in module 2). This is not an issue when we use proper integrity/authentication techniques. But when we don't, then bad stuff can happen.

Inside `utils.py`, you can find the class `ServerCTR`. This class implements a server that we can call to generate a token with our user name and password, and later we can use the token to authenticate ourselves to the server. It works as follows:

1. When instantiating an object of the server, calling `ServerCTR(seed)`, the server generates a random key (16 bytes), a random IV for CTR (8 bytes), and a random server code (24 bytes, stored as a Base64 string).

```
    def __init__(self, seed: int = 0):
        random.seed(seed)
        self.key = randbytes(AES.block_size)
        self.iv = randbytes(AES.block_size // 2)
        self.server_code =
            base64.b64encode(randbytes(24)).decode('ascii')
```

[3]We chose this song as plaintext not just because it is a bop, but because the lyrics have lots of repetition (hint).

2. Then, we can call `generate_guest_token(name, pwd)`, providing a name and a password.

```python
def generate_guest_token(self, name: str, pwd: str) -> bytes:
    name = name.replace('&','').replace('=','')
    pwd = pwd.replace('&','').replace('=','')
    token =
        f'name={name}&pwd={pwd}&role=guest&code={self.server_code}' #
        generate token
    cipher = AES.new(self.key, AES.MODE_CTR, nonce=self.iv)
    return cipher.encrypt(token.encode('ascii'))
```

   You can see that the server removes `&` and `=` characters from the name and password, and then generates a token string storing the name and password, a role (by default, `'guest'`), and the secret `server_code`. Then, the server encrypts this string using AES-CTR, and returns this encrypted token.

3. Finally, if we call `read_token(enc_token, pwd)` with the encrypted token we have received from the previous function, and the password we have used when calling the previous function, the server decrypts the token, checks that its format is correct, that the `pwd` we provided is the same one that we used when generating the token, and that the `server_code` has not been modified. If these checks pass, then the server prints the user role, as found in the decrypted token. This means we have successfully authenticated ourselves to the server.

```python
def read_token(self, enc_token: bytes, pwd: str):
    cipher = AES.new(self.key, AES.MODE_CTR, nonce=self.iv)
    token = cipher.decrypt(enc_token).decode('ascii')
    data = {kv.split('=')[0]: kv.split('=')[1] for kv in
        token.split('&')} # process the string and store the
        key-value pairs in a dictionary.

    assert 'pwd' in data
    assert 'role' in data
    assert 'code' in data
    if data['pwd'] != pwd:
        print("Incorrect password")
    elif data['code'] != self.server_code:
        print("Incorrect server code")
    else:
        print(f"Your role is {data['role']}")
```

Your goal is to call the server to generate a token (by default, this has the `guest` role), and then perform operations in the token so that, when calling `read_token(enc_token, pwd)`, the token passes all checks but your role has changed.

My recommendation is that, to do this, you start playing with the server. For example, you can implement a function in the server that decrypts a token and prints the result as a string. Then, try generating a token, printing it, modifying something in it, and printing it again. Understand how changes in the ciphertext (encrypted token) affect the plaintext (decrypted token). Use this to your advange to change the `role` field in the token.

Your solution should be in a class that you have inside `a2.py` called `AttackCTR`. We will follow this approach to grade this (take a look at the comments):

```python
def q31_attack_against_ctr():
    server = utils.ServerCTR(seed=0) # Initialize server
    attacker = a2.AttackCTR() # Initialize attacker
    # Attacker provides name and pwd
    name, pwd = attacker.generate_name_and_pwd()
    # Server generates encrypted token
    token = server.generate_guest_token(name, pwd)
    # Attacker modifies encrypted token and pwd (if needed)
    token, pwd = attacker.modify_token_and_pwd(token)
    # Server reads the modified token
    server.read_token(token, pwd)
```

Your goal is that, when the server reads the modified `token`, it prints a *different role*.

[30 pts] Question 3.1: Implement the attack `AttackCTR`, which modifies the role in an encrypted token. You will get 25 points if you change the role to `admin` (i.e., the server prints `'Your role is admin'` when calling `server.read_token`). You will get all 30 points if the server prints `'Your role is superuser'`. (This is because changing the role to a word slightly longer requires a *little* more thinking.)

Note: your attack must also work when changing the seed. You do not need to know what the secret key or `server_code` are to execute this attack. You will only call `read_token` once at the end, to check if you have changed the role successfully. You must solve this the cryptographic way; Python tricks like accessing the secret key are not allowed.

4 An attack against AES-ECB

Finally, let's see an attack against AES-ECB. Take a look at the class `ServerECB` in `utils.py`. This works like `ServerCTR`, but

1. The `ServerECB` allows `&` and `=` characters in the `name` and `pwd` variables when generating the guest token.

2. There is no `read_token` function, because we do not need it for this attack.

Your goal is to guess the `server_code` by repeatedly calling `generate_guest_token`. Turns out we do not need to learn the secret key to guess the server code!

[30 pts] **Question 4.1:** Implement the function `attack_ecb` in `a2.py`. This function receives another function as an argument: this will be the `generate_guest_token` function. By repeatedly calling this function, you must guess the `server_code`. You will get 25 points just by guessing the *first character* of the `server_code`. You have some help on how to do this below. If your attack returns the whole code instead, you will get all 30 points.

Note: as before, you cannot somehow access the variable `server_code` inside the `ServerECB` class by using Python tricks. Simply calling `generate_guest_token` with carefully-crafted `name` and `pwd` should be enough to do this question.

The evaluation would be something like this (but with changing seeds):

```python
def q41_attack_against_ecb():
    server = utils.ServerECB(seed=0) # Initialize server
    cracked_code = a2.attack_ecb(server.generate_guest_token) # Run attack
    if cracked_code[0] == server.server_code[0]:
        print("You got the first character! (25 pts)")
    if cracked_code == server.server_code:
        print("You got the full code! (30 pts)")
```

Help: cracking the first character of the code

It is recommended that you play with `ServerECB` before actually writing `attack_ecb` in `a2.py`. For example, you could add a new function in the server that takes a token, decrypts it, and prints it, block by block. Imagine we do this, and we test it:

```python
import utils
server = utils.ServerECB(seed=0)
name = 'simon'
pwd = 'password123'
token = server.generate_guest_token(name, pwd)
server.print_token_block_by_block(token) # You should know how to program this
```

This prints:

```
block 0: name=simon&pwd=p
block 1: assword123&role=
block 2: guest&code=VZSqa
block 3: zQvXQo6XkhC+rQo9
block 4: 2Lm4oLlwWV8
```

Taking the encrypted token, we can also print the ciphertext block-by-block:

```
for i in range(0, len(token), AES.block_size):
    print(f"block {i // AES.block_size}: {token[i:i+AES.block_size]}")
```

This gives a bunch of bytes objects:

```
block 0: b'FS\xdb\x13\xe8\x81\x04\xf2\x10"\x99\xe5*\x92\\\xa6'
block 1: b'l\t$\x17\xec\x9f\xf37>\x18\\\x84{\xd3Z\xd3'
block 2: b'\xe80~Bo\xfa*1\x90\xc7\xf1\x161\xd4\x9e\xa6'
block 3: b'\x97\xc6p\xab\xca\xc3\x81\xc0G\xf6\x03.\xa4\xff\xfb\xda'
block 4: b'`w\xcc\x8aON\x00Oq\x9f\xaf\x8a\xcdw&\x9e'
```

Since ECB encrypts block-by-block *with the same key*, each of the blocks of plaintext above would always be encrypted to those blocks of ciphertext below.

Now, let's try with a longer password:

```
server = utils.ServerECB(seed=0)
name = 'simon'
pwd = 'password1234567'
token = server.generate_guest_token(name, pwd)
server.print_token_block_by_block(token)
```

This prints:

```
block 0: name=simon&pwd=p
block 1: assword1234567&r
block 2: ole=guest&code=V
block 3: ZSqazQvXQo6XkhC+
block 4: rQo92Lm4oLlwWV8
```

If we print the corresponding ciphertext blocks, these are:

```
block 0: b'FS\xdb\x13\xe8\x81\x04\xf2\x10"\x99\xe5*\x92\\\xa6'
block 1: b'\x0e`\xd3\xfc\xdd\x11\xc6\x11\xacl%?\xack\x87\x04'
block 2: b'i.M\x18\xb1\xe5\xd8GA=\x7fO\xbcu<P'
block 3: b'\xd2\x19t8"\x16\x9f\x9bd\xac\xd5o\xaa~\x96\x94'
block 4: b"r{N\x7f\xc1U\x063\xed\xec\xf2_'\xd6\x91\xeb"
```

Note that we are "cheating" by printing the plaintext blocks, because we are seeing the true code; however, if we didn't cheat, we would still know the values of plaintext blocks 0 and 1, and all but the last byte (character) of block 2.

Here's the trick: if we could force the server to encrypt a plaintext block that is "`ole=guest&code=a`", then encrypt one that is "`ole=guest&code=b`", then "`ole=guest&code=c`" and so on (changing the last character, and looping through all possible ASCII characters), we would learn what the corresponding ciphertexts look like! The ciphertext block of one of those plaintext blocks (in this case, the one that ends with `V`) would be exactly the one you see as block 2 in the code block above.

So, basically, you can brute-force one byte because there are only 2^8 possible values for this byte (since you know it's ASCII, in this case is only 2^7). This is only possible because you can input any `name` and `pwd`, and because the server is using ECB mode. However, it is one of those "tricks" that have led to real-world attacks.

Start there and, if you finish and you want to get the extra 5 points, think how to extend this idea to crack the whole code. Make sure your attack works when changing the server seed. It is not hard once you crack one character, but requires more work, and it is only 5 extra points.