**Libraries to use: package Crypto** [**http://pythonhosted.org/pycrypto/**](http://pythonhosted.org/pycrypto/) **or cryptography** [**https://cryptography.io/en/latest/**](https://cryptography.io/en/latest/)

[**http://docs.python-guide.org/en/latest/scenarios/crypto/**](http://docs.python-guide.org/en/latest/scenarios/crypto/)

**All submissions will be on GitHub. The commit time stamp will be your submission time. No late submissions are allowed. Your last commit before the submission deadline will be graded.**

**Warm-up Exercises (Week 1)**

1. a) Write a program called *hex2Base64* to convert hex to base64. The hex string comes from the user.

Do not use any inbuilt functions for this conversion. Implement the conversion algorithm yourself. The program must display the base64 output on screen.

b) Write another program called *b64ConversionTester* that takes a text file as input, calls *hex2Base64* to convert it to base64 and then calls an library function (from a popular vetted library) to do the conversion and compares the result. The output must be “True” or “False”, where True implies your output matched with the one generated by the library.

The input text file will contain hex strings, one on each line (user will provide the file name as an argument when calling the program). The hex strings may contain one or more whitespaces in between characters (e.g. ea b7 5 e a 8) and/or 0x as prefix. Your program should be able to handle these. If there are multiple lines in the file, your program should print True/False for each line.

Feel free to use look-up tables/dictionary structures as needed.

Clearly state the assumptions you make on the input (length of input, etc.) and justify those assumptions (make a case why these are practical assumptions). For inputs that you think are not valid and may occur in the real world, create appropriate error handling.

Your submission must include READ ME files for both 1a and 1b detailing the usage format and any other peculiarities of your implementation.

1. XOR: Implement a function that takes two equal length hex strings and XORs them. The output must also be in hex. Print the output to the screen.

Try out:

Input a: 9c863d374184079d60066b3b4a193d3354c7

Input b: 2ca09c99b91d85d444e7c3a8beadeeff4f1c

Output: b026a1aef899824924e1a893f4b4d3cc1bdb

1. The hex string given below has been XOR’d against a single character; find the key and decrypt the message. You can use English plaintext frequencies to rank the outputs.

Decrypt:



Aside reading: <https://blog.malwarebytes.com/threat-analysis/2013/05/nowhere-to-hide-three-methods-of-xor-obfuscation/>

**Week 2: Getting Serious**

1. General XOR cipher (Vigenere Cipher): Given a key ‘ICE’, encrypt the following text.

“We didn't start the fire, It was always burning, Since the world's been turning, We didn't start the fire, No we didn't light it, But we tried to fight it”

Without the quotes.

Your output should be:

1e26652d2a2127643169303128313169372d2c632320312065630c3d6332283065282f32283a366921303b2d2c27246969102c27202069372d2c63322631292d64366921202c2d653d3637272a2b2e6f651e26652d2a2127643169303128313169372d2c632320312065630b2663322c632120272b6e3765252a222137652037696901303d63322c63313b2a202d633126632320242d3d632c3d6f65

1. Breaking Vigenere Cipher: A file is uploaded on Blackboard containing base64’d input that has been encrypted with Vigenere cipher. Your task is to decrypt it. You can use the following procedure to do so:
   1. Write a function that will compute the Hamming distance between two stings. The hamming distance is basically the number of bit positions that strings differ in. Try out your function with the following input:

is this heaven

and

no it’s iowa!!

Your computed distance should be 46. *This needs to be correct.*

Determine the keyLength:

1. Make a guess for a keyLength. You can try different values 1 to 45.
2. For each keyLength that you guess (1 to 45), divide the input ciphertext into chunks of that size and compute the Hamming distance between adjacent chunks. Example, if you are trying keyLength = 4, then divide the ciphertext into chunks of 4 bytes and compute the Hamming distance between adjacent chunks. Normalize each Hamming distance by keyLength currently being tried and then take the average, in the end for that keyLength. You will build up a dictionary of keyLength and corresponding Hamming distance.
3. The smallest normalized Hamming distance will most probably be the actual keyLength.

Break the encryption:

1. Now that you know the most probable key size: break the ciphertext into blocks of keyLength.
2. Now create new blocks composed of: first byte of each block, 2nd byte of each block, 3rd byte of each block, etc.
3. Solve each of these new blocks as if they were XOR’d with a single character (problem 3 from last week). This will give you the key byte for that block.

This code is extremely error prone so be patient while debugging. The code will be quite useful in later problems.

**Week 3: Block Ciphers**

**Libraries to use: PyCrypto:** [**http://pythonhosted.org/pycrypto/**](http://pythonhosted.org/pycrypto/) **or cryptography:** [**https://cryptography.io/en/latest/**](https://cryptography.io/en/latest/)**. I use cryptography.**

Which library is compatible with which version of Python: [**http://docs.python-guide.org/en/latest/scenarios/crypto/**](http://docs.python-guide.org/en/latest/scenarios/crypto/)

1. AES in ECB mode: give different encrypted text with key “NO PAIN NO GAIN!”

This takes maybe 30 minutes to 1 hour

1. Detect AES in ECB mode: Replicate Matasano’s file with different text and a twist

1-2 hours

1. Implement PKCS#7 padding for general block length. Give a random message and ask to pad to 160 bit block.

I took the sentence “This is a Saturday” and the output was: 'This is a Saturday\x02\x02'

1. **Implement CBC mode:** Implement CBC mode by hand using the ECB function you wrote earlier. It must encrypt and decrypt text. And use the XOR function from the previous exercise to combine them. Don’t use the built-in CBC mode.

Although the library has a CBC mode and that’s what you *must* use in real world implementations, the point of this exercise is to make you carefully look into the CBC mode and really understand it by doing it rather than just reading the theory. We will use this understanding later to break some stuff.

**Week 4: Decrypting secret cookies encrypted using ECB mode**

1. An ECB/CBC detection oracle: <https://vimeo.com/41116595>

<http://pentest.cryptocity.net/>

<https://news.ycombinator.com/item?id=7959519>

Using your existing ECB and CBC code, do the following.

* Generate a random AES key
* Write a function that encrypts data under the unknown AES key you just generated. Use any input you like to encrypt.

You now have an encryption oracle (much like a Web Server). You give it an input and it gives you encrypted stuff.

* Now the encryption function secretly also does the following: prepend random 5-10 bytes and append random 5-10 bytes to the plaintext.
* Further, the encryption function must randomly choose which mode of encryption to follow ECB or CBC. Use a random IV for CBC – it does not matter.

Given the above encryption oracle, implement a function that detects the block cipher mode. So in the end you have a function that when given a randomly encrypted text, under an unknown key and IV will tell you whether the mode used was ECB or CBC.

**Question:** *Sometimes ECB mode may go undetected (assuming your code is working fine). Why do you think it happens?*

1. Byte-at-a-time ECB decryption (Simple): **2 hours**

You can get rid of the dictionary as well that they ask you to create

**Week 5: Block Ciphers Continued**

1. PKCS#7 padding validation

Make the function throw an exception for invalid pads

This is needed later for CBC padding oracle attack

1. CBC bitflipping attacks

Will take one-week probably

**Clean up week: Eliminate code repetition, make for loops shorter more efficient, make sure there are no magic numbers, make sure everything is commented, recompile every program and update their executables, update read me files. Document all the clean-up you did for every program in comments on top of the program. Document dependencies (which module calls which module)**

**Week 6: Block Ciphers Continued**

1. The CBC padding oracle attack

Will take one-week for sure

1. Implement CTR mode

**Week 7: Stream Ciphers**

1. Implement Mersenne Twister RNG
2. Crack a MT19937 seed
3. Clone an MT19937 RNG from its output
4. Create the MT19937 stream cipher and break it

**Week 8: Stream ciphers continued**

1. Break "random access read/write" AES CTR
2. CTR bitflipping
3. Recover the key from CBC with IV=Key – find a real world example of this error for context

**Week 9: Hash functions**

1. *Implement* SHA-1 keyed MAC – write your own code
2. Break a SHA-1 keyed MAC using length extension
3. Break an MD4 keyed MAC using length extension

Week 9: