**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.

**Week 3: Block Ciphers**

**Crypto Libraries to use:**

* PyCrypto: <http://pythonhosted.org/pycrypto/> - for older versions of python
* pycryptodomex: <http://www.blog.pythonlibrary.org/2016/05/18/python-3-an-intro-to-encryption/> - for new versions of python
* cryptography: <https://cryptography.io/en/latest/>

I used cryptography for all the symmetric key algorithms. But then found their documentation for public key crypto quite bad. So now for public key crypto I am switching over to pycryptodomex. Maybe you want to start with pycryptodomex from the beginning.

A reference blog: <http://www.blog.pythonlibrary.org/2016/05/18/python-3-an-intro-to-encryption/>

1. Decrypt the base64 encoded ciphertext given in w3p1.txt. AES-128 in ECB mode was used for encryption and the key used was: “NO PAIN NO GAIN!”

Feel free to use one of the above python crypto libraries for AES. Yes, this one is actually that simple!

1. Each line of the file w3p2.txt contains AES encrypted ciphertext. One of them uses ECB mode. Your code should detect which one and print that line number.
2. Implement the PKCS7 padding for AES.

For example for, if I want to PKCS7 pad the phrase “This is a Saturday” to 160 bits block length, the output is: 'This is a Saturday\x02\x02'

What is the padding if your plaintext is “NO PAIN NO GAIN!”? Give me the output and an explanation.

This observation will be very useful in when we try to decrypt improperly encrypted cookies in a few weeks (determining block length).

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

1. **AES CBC mode:** Implement CBC mode by hand using the ECB function you wrote earlier (problem 1). It must encrypt and decrypt text. You will need to XOR blocks but you already have a function for that from the previous exercises that you can combine. *Don’t use the built-in CBC mode.*

Decrypt the file w3p4.txt using the key ‘NO PAIN NO GAIN!’ and an IV of all 0 bytes.

Although all the crypto libraries have 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.

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. Decrypting ECB encrypted data – Part 1

Reuse the ECB encryption oracle function you wrote in the previous problem and modify it so that it encrypts under a fixed unknown key. Don’t change the key again and again.

Before encrypting any user input, the encryption function must append the following to the plaintext (available in p11.txt):

**Warning:** Do ***NOT*** read the base64 decoded string. Make your code decode it for you and use it right away.

V2hlbiBJIHdhcyBhIGxpdHRsZSBiaXR0eSBiYWJ5DQpNeSBtYW1hIHdvdWxkIHJvY2sgbWUgaW4gdGhlIGNyYWRsZSwNCkluIHRoZW0gb2xkIGNvdHRvbiBmaWVsZHMgYmFjayBob21lOw0KSXQgd2FzIGRvd24gaW4gTG91aXNpYW5hLA0KSnVzdCBhYm91dCBhIG1pbGUgZnJvbSBUZXhhcmthbmEsDQpJbiB0aGVtIG9sZCBjb3R0b24gZmllbGRzIGJhY2sgaG9tZS4=

You shouldn’t know the contents of this string. This is the unknown part that you may find in a cookie. The input to the encryption looks like:

AES\_128\_ECB(attacker\_controlled||unknown\_string, random-key)

You will attempt decrypt it by repeatedly calling the encryption oracle (which is the server) without actually knowing the key. Here is what you’ll do:

1. **Find the block size:** Feed the encryption oracle one byte, say ‘X’, at a time. So first time you feed it ‘X’, next time you feed it ‘XX’ and so on. Look at the output you get back. What you are looking for is a jump in the size of the ciphertext. Think what does the jump indicate (think padding!)
2. **Detect the encryption mode:** Use your previous ECB detection function to detect if the cipher is using ECB mode.

Now that you know the block size and the encryption mode, start decrypting the unknown string byte at a time.

1. Craft two identical blocks that are one byte shorter than the block length. So if the block size was 6 bytes, you would create two blocks containing “XXXXX”. Now for the last byte of the first block you are going to try out all the different possible byte values. So you start with XXXXX0, XXXXX1 and so on.

Let’s say in the first iteration you have XXXXX0 for the first block, you concatenate the second block and get XXXXX0XXXXX. You feed this to the encryption function and pass the encrypted output through ECB detection oracle. If it detects an ECB mode then it means that the first byte of the unknown string is 0.

You may have to think a little on what the encryption oracle is doing.

If ECB is not detected continue to XXXXX1 and so on.

1. Once you have the first byte, try decrypting the second byte and so on until the entire unknown string is decrypted.

**Week 5: Block Ciphers Continued**

1. Decrypting ECB encrypted data – Part 2

Take your code from problem 11 and modify the encryption oracle such that not only adding the target\_bytes after the attacker\_controlled input it also adds a random number of bytes to the start (between 10-20 bytes). Generate these bytes randomly. So the encryption function basically does this:

AES-128-ECB(random\_bytes||attacker\_controlled||target\_bytes, random\_key)

Like before, your goal is to decrypt the target\_bytes. Find the target\_bytes for this problem in p12.txt file.

1. PKCS#7 padding validation

Write a function that takes a plaintext, checks if it has a valid PKCS #7 padding. If the padding is valid, your function must strip off the padding.

Example:

The string: 'This is a Saturday\x02\x02' has a valid padding

And your function will produce an output: 'This is a Saturday'

The string: 'This is a Saturda\x03\x02\x02' does not have a valid pad, nor does 'This is a Saturda\x03\x02\x01'

Your function must throw an exception for invalid pads

You will play with padding quite a bit in a later problem. So know it inside out.

**Week 6: Block Ciphers Continued**

1. CBC bitflipping attacks

<http://resources.infosecinstitute.com/cbc-byte-flipping-attack-101-approach/>

<http://swepssecurity.blogspot.com/2014/05/bypassing-encrypted-session-tokens.html>

<http://blog.gdssecurity.com/labs/2010/10/6/crypto-challenges-at-the-csaw-2010-application-ctf-qualifyin.html>

Will take one-week probably

**Clean up week: Eliminate code repetition, make loops shorter/more efficient, make sure the magic numbers are there for a good reason and clearly marked and easy to find, make sure everything is commented, recompile every program and make sure it works, 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), etc.**

**Week 7: Block Ciphers Continued**

1. The CBC padding oracle attack

Will take one-week for sure

1. Implement CTR mode

**Week 8: Stream Ciphers**

1. Break fixed-nonce CTR mode using substitutions
2. Break fixed-nonce CTR statistically

**Week 9: 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 10: 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 11:

1. Implement and break HMAC-SHA1 with artificial timing leak
2. Break HMAC-SHA1 with slightly less artificial timing leak

**Week 12: Public Key Algorithms**

1. Implement Diffie-Hellman

Set a variable p=37 and g=5. Generate ‘a’, a random number mod 37 (remember to use randomness from os.urandom (<https://cryptography.io/en/latest/random-numbers/>) and not from random()). You can also use secrets.py in version 3.6 to generate random numbers.

Now generate “A”, which is “g” raised to the “a” power mod 37: A=g\*\*a mod p.

Do the same for “b” and “B”

“A” and “B” are public keys. Generate a session key with them: set “s” to “B” raise to the “a” power mod 37: s=B\*\*a mod p.

Do the same with A\*\*b, check that you come up with the same “s”.

To turn “s” into a key, you can just hash it to create 128 bits of key material (or SHA256 it to create a key for encrypting and a key for a MAC).

Now repeat the exercise with bignums like in the real world. Here are parameters NIST likes:

p:

ffffffffffffffffc90fdaa22168c234c4c6628b80dc1cd129024

e088a67cc74020bbea63b139b22514a08798e3404ddef9519b3cd

3a431b302b0a6df25f14374fe1356d6d51c245e485b576625e7ec

6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5ae9f

24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361

c55d39a69163fa8fd24cf5f83655d23dca3ad961c62f356208552

bb9ed529077096966d670c354e4abc9804f1746c08ca237327fff

fffffffffffff

g: 2

Note that you'll need to write your own modexp (this is blackboard math, don't freak out), because you'll blow out your bignum library raising "a" to the 1024-bit-numberth power. You can find modexp routines on Rosetta Code for most languages.

**Note:** *Do not make any global variables for this exercise.*

1. Implement Secure Remote Password (SRP)

They will run into exponentiation taking very long time. They will need to use trick from number theory to make it work like doing mod often before numbers grow too large.

Implement algorithm from here: <http://srp.stanford.edu/design.html>

REPL on client side. Server says go or no-go.

1. Break SRP with a zero key

Client sends A = 0 and sets S=0 for itself. This lets the client login with any username and password. Let’s students figure out what client should do (S=0) to login successfully. What safeguards must the server implement to prevent this.

Week 13:

1. ~~Offline dictionary attack on simplified SRP~~
2. Implement RSA

Make them implement invmod function. This is useful for next problem.

1. Implement an E=3 RSA broadcast attack

Week 14:

1. Implement unpadded message recovery oracle
2. Bleichenbacher’s e=3 RSA attack

Week 15:

1. DSA key recovery from nonce
2. DSA nonce recovery from repeated nonce
3. DSA parameter tampering

Week 16: (Final Week)

1. Bleichenbacher's PKCS 1.5 Padding Oracle (Simple Case)
2. Bleichenbacher's PKCS 1.5 Padding Oracle (Complete Case)

**Bonus Problems:**

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