Crypto Report

Heytem BOUMAZA
11926806

M1 Informatique 2019-2020

Contents

Section I

Qualifying Set

I.1 Convert Hex to Base64

I.1.1 Solution description

This one is pretty straightforward, using the commons.codec package from apache, we can do conversions from and to both Hex and Base64, with simple calls to endcoding and decoding methods.

I.1.2 Code

```
import org.apache.commons.codec.DecoderException;
import org.apache.commons.codec.binary.Base64;
import org.apache.commons.codec.binary.Hex;

public class HexToBase64 {

        //convert from Hex to Base64
        public static String FromHex_ToBase64(String hex_str )
        throws DecoderException{

        byte[] decodedHex = Hex.decodeHex(hex_str);
        return Base64.encodeBase64String(decodedHex);

        }
        //convert from Base64 to Hex
        public static String FromBase64_ToHex(String base64_str) {
```

```
        byte[] decodedBase64 = Base64.decodeBase64(base64_str);
        return Hex.encodeHexString(decodedBase64);

        }
```

I.1.3 Code Execution

Main

```
public static void main(String[] args) {
        String hex_buffer1 = "49276d206b696c6c696e6720796f7572206
        27261696e206c696b65206120706f69736f6e6f7573206d757368726f6f6d";
        try {
        //Hex to Base64 conversion
        System.out.println(FromHex_ToBase64(hex_buffer1) );
        //the encoded String content
        System.out.println(new String (Base64.decodeBase64
        (FromHex_ToBase64(hex_buffer1))));


        }catch(DecoderException e) {
                System.out.println(e.getMessage());

        }

        }
```

Output

```
SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
I'm killing your brain like a poisonous mushroom
```

I.2 Fixed XOR

I.2.1 Solution description

As you can see below in the code section, the method fixed_xor takes two Hex encoded strings, recovers their byte arrays, loops through them and performs single byte xor operations and append the result to a new byte array then returns it hex-encoded.

I.2.2 Code

```
import org.apache.commons.codec.*;
import org.apache.commons.codec.binary.Hex;

public class FixedXor {


public static String fixed_xor(String buffer1,String buffer2) {
    try{
        byte[] buffer1_bytes = Hex.decodeHex(buffer1);
        byte[] buffer2_bytes = Hex.decodeHex(buffer2);
        byte[] xor_result = new byte[buffer1_bytes.length];

        for(int i=0;i<buffer1_bytes.length;i++) {
        xor_result[i]= (byte)(buffer1_bytes[i] ^ buffer2_bytes[i]);
                }

                return Hex.encodeHexString(xor_result);

            }catch(DecoderException e) {
                System.out.println(e.getMessage());

            }

                return null;


        }
```

I.2.3 Code Execution

Main

```
public static void main(String[] args) {
  String buffer1 ="1c0111001f010100061a024b53535009181c";
  String buffer2 ="686974207468652062756c6c277320657965";
  System.out.println("result: "+fixed_xor(buffer1,buffer2));
   try {
        System.out.println(new String(Hex.decodeHex
        (fixed_xor(buffer1,buffer2))));
        } catch (DecoderException e) {

        e.printStackTrace();
        }
```

Output

```
result : 746865206b696420646f6e277420706c6179
the kid don't play
```

I.3 Single-Byte XOR cipher

I.3.1 Solution description

The first thing i tackled in this challenge was finding out how to properly compute the score based on character frequencies : given the frequency of each character of a certain text, how should this text be scored ?. To my understanding, a score represents how "far" a text is from being a valid English text or just gibberish, the obvious way to do it is to compute the difference between the actual character frequency and the observed frequency then put them in some mathematical formula to get the score, and thanks to "Mathematics meta stackexchange" forums, the following formula worked like charm for me : $X2 = \sum c(Nobs(c) - Nexp(c))2/Nexp(c)$, it's called chi-squared test, it gives the sum of the squared differences between the observed and the actual frequency, divided by the actual frequency of the letter in the english language. The smaller the score, the more this text resembles English, it turned out this formula has many applications and not just in cryptography, such as bioinformatics and data science.

the code is divided into these methods ;

- **SingleByteXor** : this method performs the xor operation of every possible character against the message and stores the decryptions in a hashmap with their corresponidng keys, then send it to the method "getTextWithBestScore".
- **getTextWithBestScore** : this method takes the hashmap containing the decryptions and computes the score for each entry by calling "computeTextScore", it returns the decryption with the best score.
- **computeTextScore** : takes the hex encoded decryption as well as a hashmap containing the frequencies of the english letters, makes call to getMessageLettersFrequencies to get a dictionary of frequencies for each letter of the specified the decryption, after that it computes the score using chi-squared test.

- **getMessageLettersFrequencies** : takes byte array of a message and returns a dictonary containing the frequency of each English letter in the message.
- **getLettersFrequencies** :returns a hashmap containing the frequencies of letters in the english language.
- **duplicateKey** : because we need to xor a single byte against a byte array and the fixedxor method only accepts buffers of equal length, this method duplicates the byte key to be the same length as the message.

I.3.2 Code

Check the class named 'SingleByteXorCypher' in the source files

I.3.3 Code Execution Result

Main

```
public static void main(String[] args) throws DecoderException {
 ArrayList<String> result = new ArrayList<String >();

 String encrypted_message="1b37373331363f78151b7f2b783431333
 d78397828372d363c78373e783a393b3736";

 result=SingleByteXor(encrypted_message);

try {
 System.out.println("hex-encoded message :"+new String(result.get(1)));
 System.out.println("message : "+new String(Hex.decodeHex(result.get(1))));
 System.out.println("the key is : "+result.get(0));

}catch(DecoderException e) {

}

}
```

Output

```
hex-encoded message :436f6f6b696e67204d432773206c69
6b65206120706f756e64206f66206261636f6e
message : Cooking MC's like a pound of bacon
```

```
the key is : X
```

I.4 Detect single-character XOR

I.4.1 Solution description

This challenge is an extension of the previous one, the most obvious solution is to perform a single-byte-xor brute force on each line and pick the best score.

I.4.2 Code

Check the class named 'DetectSingleCharacterXor' in the source files

I.4.3 Code Execution Result

Output

```
6e4f570054484154005448450050415254590049530004a554d50494e472a
true message :nOW THAT PARTY IS JUMPING*
```

I.5 Implement repeating-key XOR

I.5.1 Solution description

One way to do this challenge is to duplicate the key bytes in a sequential manner until its length reaches the length of the message (at least), but the way i chose to do it is to use an index on the key bytes that resets to 0 every time it reaches the last byte of the key, this approach was easier to implement.

I.5.2 Code

```
public static String RepeatKeyXorCypher(String message,String key) {
byte[] message_bytes = message.getBytes();
byte[] key_bytes = key.getBytes();
int j=0;
byte[] encrypted_message_bytes=new byte[message_bytes.length];
//loop through the message bytes
for(int i=0;i<message_bytes.length;i++) {
encrypted_message_bytes[i]=(byte)(message_bytes[i]^key_bytes[j]);
        j++;
```

```
 //reset the index bytes if it reaches the key length
 if(j==key_bytes.length) {
         j=0;
  }

}

 return Hex.encodeHexString(encrypted_message_bytes);



}
```

I.5.3 Code Execution Result

Main

```
public static void main(String[] args) {
        String message ="Burning 'em, if you ain't quick and nimble
        I go crazy when I hear a cymbal";
        String key ="ICE";
        System.out.println(RepeatKeyXorCypher(message,key));
        }
```

Output

```
0b3637272a2b2e63d72c2e69692a23693a2a3c6
324202dd73d63343c2a26226324272765272a282
b2f2044490c69242a69203728393c69342d2c2d65
00632d2c22376922652a3a282b2229
```

I.6 Break repeating-key XOR

I.6.1 Solution description

This is by far the challenge i struggled the most with, mainly because i didn't know what i was doing from step 2 to 6, even though the technique to find the length of the key is given step by step, how and why it works wasn't obvious to me so i had to do some research on the theoretical background of this technique, turned out what we were trying to do all along was grouping together ciphertext bytes that share the same key byte. unfortunately, i kept getting incorrect results every time.

I.6.2 Code

Check the class named 'BreakRepeatingKeyXor' in the source files

I.6.3 Code Execution Result

Main

```
public static void main(String[] args) {
String s1 ="this is a test";
String s2="wokka wokka!!!";
//System.out.println("hamming distance = "+
computeHammingDistance(s1.getBytes(),s2.getBytes()));
//expected output : 37 ;
String url="src\\crypto\\repeating-xor.txt";

breakRepeatingKeyXor(getTheKeySize(url),readEncryptedFile(url));



            }
```

Output

I kept getting weird results :(

I.7 AES in ECB mode

I.7.1 Solution description

To perform AES in ECB mode encryption and decryption operations, i'm going to use the 'crypto' package in javax :

- 1. First we instantiate a cipher object and pass the encryption and decryption mode as a parameter, in our case it's AES in ESB mode
- 2. We instantiate a key object with the cipher mode and the key byte array passed as parameters to the constructor
- 3. We initiate the cipher by calling 'init' and passing the key instance and the mode number as parameters
- 4. We perform the encryption/decryption by calling 'doFinal' on the cipher object and recover the result.

I.7.2 Code

```
//Encryption method
public static byte[] encryptAesInEcb(byte[] key , byte[] message) {

                try {
                Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
                //create the AES key
                Key aes_key = new SecretKeySpec(key,"AES");

                //decrypt mode
                cipher.init(Cipher.ENCRYPT_MODE, aes_key);

                //decrypt the message
                byte[] decrypted_message=cipher.doFinal(message);

                 return decrypted_message;

                } catch (NoSuchAlgorithmException |
                NoSuchPaddingException e) {
                System.out.println(e.getMessage());
                } catch (InvalidKeyException e) {
                System.out.println(e.getMessage());
                e.printStackTrace();
                } catch(IllegalBlockSizeException |
                BadPaddingException e) {
                System.out.println(e.getMessage());
                }
                                return null;


        }

//Decryption method
public static byte[] decryptAesInEcb(byte[] key,byte[] message) {
        try {

        Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
        Key aes_key = new SecretKeySpec(key,"AES");
```

```
//decrypt mode

cipher.init(Cipher.DECRYPT_MODE, aes_key);

return cipher.doFinal(message);

} catch (NoSuchAlgorithmException | NoSuchPaddingException e) {
System.out.println(e.getMessage());
e.printStackTrace();
} catch (InvalidKeyException e) {
System.out.println(e.getMessage());
e.printStackTrace();
} catch(IllegalBlockSizeException | BadPaddingException e) {
System.out.println(e.getMessage());
        }
        return null;


}
//Reading the content of the file
public static byte[] readEncryptedFile(String url) {
        {
                try {
                File file=new File(url);
                FileReader fr=new FileReader(file);
                BufferedReader br=new BufferedReader(fr);
                StringBuffer sb = new StringBuffer();
                String line ;
                String text="";

                while((line=br.readLine())!=null)
                {
                sb.append(line);
                sb.append("\n");
                text = text+line;

                }
```

```
                        byte[] content_bytes = null ;
                        //base64-decode the text
                content_bytes = Base64.decodeBase64(sb.toString());
                                fr.close();
                                return content_bytes;
                        }catch(FileNotFoundException e) {
                                System.out.println(e.getMessage());

                        } catch (IOException e) {
                                System.out.println(e.getMessage());
                        }

                        return null;


        }
        }
```

I.7.3 Code Execution

Main

```
public static void main(String[] args) {

        String key ="YELLOW SUBMARINE";
        byte[] encrypted_message = readEncryptedFile("C:\\Users
        \\HAITAM\\Desktop\\crypto\\aes-in-ecb.txt");
        System.out.println(new
        String(decryptAesInEcb(key.getBytes(),encrypted_message)));

        }
```

Output

```
I'm back and I'm ringin' the bell
A rockin' on the mike while the fly girls yell
In ecstasy in the back of me
Well that's my DJ Deshay cuttin' all them Z's
Hittin' hard and the girlies goin' crazy
Vanilla's on the mike, man I'm not lazy.
.........
```

I.8 Detect AES in ECB mode

I.8.1 Solution description

We know that AEC in ECB mode uses the same key when encrypting/decrypting blocks of data, as a result, if there was two identical blocks in the original plaintext then the encryption algorithm will produce two identical ciphertexts.
So basically the ciphertext with the most duplicated blocks is most probably the one that has been encrypted under ECB mode.
The code below is divided into these methods :

- **readEncryptedFile :** this method reads a file line by line, stores them in an arrayList and returns it.
- **detectECB :** this is the method takes the arrayList that contains the ciphertexts, splits each ciphertext into blocks of equal length (16 bytes), then counts the number of duplicate blocks in each ciphertext by calling 'countDuplicateChunks', returns the one with the highest number of duplicates.
- **countDuplicateChunks :**takes as a parameter a ciphertext and returns the number of duplicate blocks.
- **alreadyCounted :** just a utility method that checks if a hashmap contains a certain value.

I.8.2 Code

Check the class named'DetectAesInEcb' in the source files

I.8.3 Code Execution

Main

```
public static void main(String[] args) {
                // TODO Auto-generated method stub
                String url ="src\\detect-aes-ecb.txt";
                detectECB(url,16);
        }
```

Output

```
ciphertext : d880619740a8a19b7840a
8a31c810a3d08649af70dc06f4fd5d2d69c744cd28
3e2dd052f6b641dbf9d11b0348542bb5708649af70
dc06f4fd5d2d69c744cd2839475c9dfdbc1d46597949d
9c7e82bf5a08649af70dc06f4fd5d2d69c744
cd28397a93eab8d6aecd566489154789a6b0308649af7
0dc06f4fd5d2d69c744cd283d403180c98c8f6
db1f2a3f9c4040deb0ab51b29933f2c123c58386b06fba186a
```

Section II

Block Crypto

II.1 Implement PKCS#7

II.1.1 Solution description

The challenge is purely technical, the padding size is computed by multiplying the desired block size by the result of (length of the text / block size)+1, this way the block will be padded correctly whether its length is less or more than the specified block size, however, if the block is equal to the specified block size i'm still going add an extra block of padding.

II.1.2 Code

Check the class named 'PKCS$_padding'inthesourcefiles$

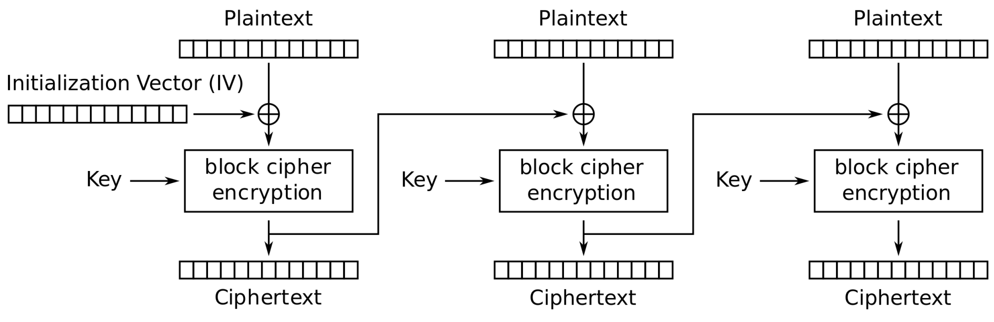
Output

```
padding size :4
after padding : 59454c4c4f57205355424d4152494e4504040404
length before padding : 16
length after padding : 20
```

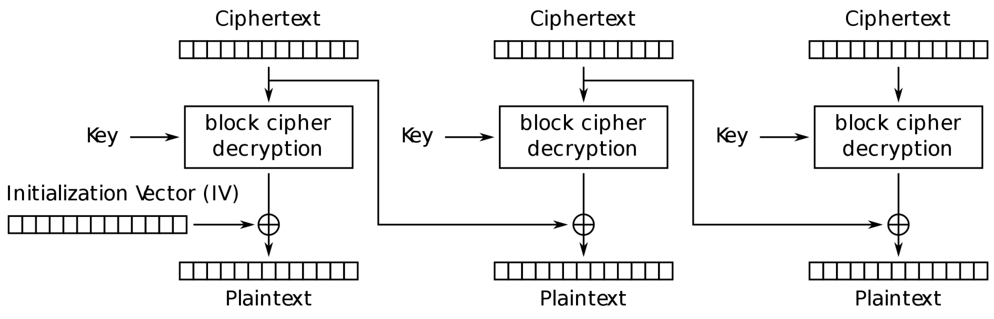
II.2 Implement CBC mode

II.2.1 Solution description

This challenge can also be considered purely technical given the fact that i'm already familiar with CBC mode encryption and decryption schemes, they can be best illustrated with the following circuits :



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

II.2.2 Code

Check the class named 'CBCMode' in the source files

II.2.3 Code execution

Output

I'm back and I'm ringin' the bell A rockin' on the mike while the fly girls yell In ecstasy in the back of me Well that's my DJ Deshay cuttin' all them Z's Hittin' hard and the girlies goin' crazy Vanilla's on the mike, man I'm not lazy.
I'm lettin' my drug kick in It controls my mouth and I begin To just let it flow, let my concepts go My posse's to the side yellin', Go Vanilla Go! ....

II.3 An ECB\CBC detection oracle

II.3.1 Solution description

For this challenge i had to modify my code several times, especially to keep count of the prediction accuracy of the oracle function. Here is how i did it: the encrytpion mode is determined randomly and performed on the plaintext using the method 'secretCipher' then the result of the cipher as well as an integer value (0 or 1) indicating the mode that has been used are sent to the oracle, the oracle function called 'encryptionOracle' looks for duplicates in the ciphertext, returns true if its prediction matches the used mode (passed as a parameter) or false otherwise, in the main method we try to feed input to the oracle method several times and keep track of its performance, further discussion of the results after the code section.

II.3.2 Code

Check the class named 'EcbCbcOracle'in the source files

II.3.3 Code execution

Output

```
left pad : 6
right pad :5
input size after the random padding :51
padding size :13
input size after the pkcs padding 64
Attempt number : 0
mode used is CBC
guess : CBC
Attempt number : 1
mode used is CBC
guess : CBC
Attempt number : 2
mode used is CBC
guess : CBC
Attempt number : 3
mode used is CBC
guess : CBC
Attempt number : 4
mode used is ECB
```

```
guess : CBC
Attempt number : 5
mode used is CBC
guess : CBC
Attempt number : 6
mode used is CBC
guess : CBC
Attempt number : 7
mode used is ECB
guess : CBC
Attempt number : 8
mode used is ECB
guess : CBC
Attempt number : 9
mode used is CBC
guess : CBC
Attempt number : 10
mode used is ECB
guess : CBC
Attempt number : 11
mode used is CBC
guess : CBC
Attempt number : 12
mode used is CBC
guess : CBC
Attempt number : 13
mode used is CBC
guess : CBC
Attempt number : 14
mode used is ECB
guess : CBC
Attempt number : 15
mode used is ECB
guess : CBC
Attempt number : 16
mode used is ECB
guess : CBC
Attempt number : 17
mode used is CBC
```

```
guess : CBC
Attempt number : 18
mode used is ECB
guess : CBC
Attempt number : 19
mode used is ECB
guess : CBC

oracle was right 11 times.
percentage of correct guesses 55.0
input size before the random padding :40
input size after the full padding 64
```

II.3.4 Discussion

Perhaps the idea behind this challenge is to simulate an attacker that controls the input and wants to know if the ECB mode is being used or not, when feeding the oracle messages like "AAAAAAAAAA.." the prediction accuracy increases until it reaches 100 percent when the input size is equal or greater than 48 bytes (before the padding occurs), this makes sense, assuming that each mode has 50 percent chance to be used over the other and that the bytes appended to the plaintext are random : for the oracle to be able to detect ECB, it must find at least 2 identical blocks in the ciphertext, that means 32 bytes if the block size is 16 which it is in our case. Worst case scenario is when we append 10 random bytes before and after the plaintext, each random padding should take 6 bytes of the plaintext to complete the block size, so at the very least we need a plaintext of size 32 + 16 = 48 bytes to guarantee 100 percent prediciton accuracy, the 48 byte plaintext according to my code will be pkcs-padded before feeding it to the oracle, which brings its size to 64 bytes total .

II.4 Byte-at-time ECB decryption

II.4.1 Solution description

First step is to find the block size, we do this by continuously feeding data to the modified oracle function and observing the size of the resulting message, if it changes that means we reached the block size limit which indicates that an extra padding block has been added, the amount of bytes added is the block size. Once we recovered the block size, we follow the steps described in the challenge to crack the ciphertext byte by byte :

- we craft our input
- move one byte ahead each time

- feed the ciphertext
- repeat for all the 256 characters
- store the results in a hashmap, this is our dictionary
- feed the oracle method the ciphertext and test it with all the elements of the dictionary, append the the matching element to the result plaintext
- repeat the process ’block-size’ times

II.4.2 Code

Check the class named ’ByteAtTime’ in the source files

II.4.3 Code execution result

Output

```
previous block size :144
current block size :144
previous block size :144
current block size :160
ciphertext size jumped by an extra block which means
block size  : 16
size of our input  : 6
the secret text :
Rollin ’ in my 5.0
With my rag—top down so my hair can blow
The girlies on standby waving just to say hi
Did you stop? No, I just drove by
```

II.5 PKCS7 padding validation

II.5.1 Solution description

This is also a technical challenge : we loop through the message, recover the last byte (the padding byte), loop through the message padding-byte-int-value times and compare the two bytes.

II.5.2 Code

Check the classe named ’ValidatePkcspadding’ in the source files

II.5.3 Code execution result

Output

```
padding size :4 after padding : 59454c4c4f57205355424d4152494e4504040404 after stripping
the padding : 59454c4c4f57205355424d4152494e45
```