Sami Noor Syed
Cyber Security
OSU Fall 2023
10/18/2023

Programming Project 1
3.1-3.3 and 3.5-3.7 & Tasks 3.4 and 3.8

Note:
The code for tasks 3.4 and 3.8 are included in the accompanying zip file

3.1:

Observation: this seems about right, each of the encrypted lines is at least as long as the original message if not longer
Plaintext: “Well here's some text that I need to encrypt... sorry it's not some clever quote :/”
Ciphertext (cleared formatting):

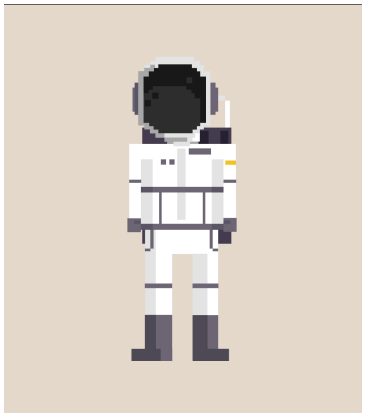
1) Aes-128-cbc:
    a) ^@<83>}¢Ê§/§<87>l|q<9f>Ù8û°<9d>¹+Â'sP^<8f>*.y®q²<8c>á<8f>iëç^E<89>^\åÉP5^A½~$gw^EØ<9b>ÆÆm^KJl"^W!E^Õ¶Ëf<^V\-<94><¥ñð¹<84>-^D¯Í´®èí^CÁZ¿¼^A^H
2) Aria-128-cfb:
    a) <87>w³^FÚ>Û*i<98>^Q|<9a>-òä@^SÈØ<80>^K~<87>8ô3=6<87>:½Dªì¢¢zqî;×^U<98><90>5<95><95>`r^T^L¾ù<80>G;4"æä§HÜlá<98>ÓÊ^E,'×uA W"ÖB^@l.

3) Aes-128-ecb:
    a) <87>ÇÌò¾W¼\<8b>þ,^A<9f>TÞ[<81>¡8"¡^\ljo`¾^Q£ñT?^M;ä^ZqÞ@H#Ò<9d>Xfl^P^@<9d>®<8e>Á*ël<93>^VÈÉ<85>â^Q¹,©^^<80>!^XÄ¨MØAÊ·un!^M1ÞÜ"<9d>¾±<9e>Û^MæÞ^T

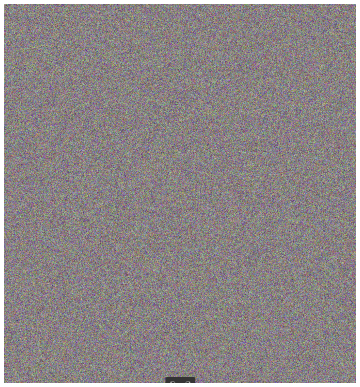
3.2:

Observation: I really enjoyed the ECB image in this one! While CBC completely scrambles the image, in ECB the image is very easy to make out. This is because it uses the same algorithm on uniformly sized blocks of information from the original code. While we may not know what the original colors were, we can definitely make out the shape of the data as we have uniform encryptions on blocks of information much smaller than the original data set.

Original:



CBC:



ECB:

Image attribution: "https://www.freepik.com/free-vector/astronaut_2921422.htm#query=bmp&position=10&from_view=keyword&track=sph" on Freepik

3.3:

1) The contents of encrypt 1 and 2 match as they should. To be able to decrypt these messages, there has to be consistency in the way that messages are encrypted. If we encrypt a file twice using the same key and the same initialization vector, then we need to be able to decrypt that file with the same information to get the same result
2) It makes sense that using a different initialization vector would result in a different encrypted message especially since chaining block ciphering involves performing an xor with the initial plaintext. The resulting ciphertext is then xored with the next bit of plaintext to produce new cipher text. In this way, a different initialization vector should change your entire cipher text.

3.4:

My program identified the key as the word “median” please see the read me in the attached files. For directions about how to run the program please see the attached README.md inside of the exercise_3.4 directory

3.5:

Using computed hashes seems like an interesting methodology of letting end receivers know that their message may or may not have been tampered with. The hashes all seem shorter than the original message,which I think makes sense.

PlainText:
And we have another plaintext file, but this time we're using hashes to encrypt... right?

MD5 :
bdbae5750e5d8aae442a29aa21be7a15

Sha-256:
fe9e2fb6969106a62941531c414cdd80eac70f41f1a7cf2ec60cf43869b9860a

Sha-1:
3367ad397869dfb20b98273c8697886da6d31cf1

3.6:

1) It seems that the key size can be any length with HMAC, but with a little online snooping, I found out that Microsoft recommends that it should be 64 bytes long for the sha256 presumably to increase the resilience of the key in the face of

attacks. I believe the key size depends on which cryptographic algorithm you use and what your individual security requirements are. Microsoft outlines later in the below link that a key that matches or exceeds the output size of the underlying hash function is often appropriate. Using a longer key provides extra security. (https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha256.-ctor?view=net-7.0)

3.7:

1) Looking at the below hash codes with your eyes does not reveal much of a pattern, but I'll list under the bit flipped column the details of using excel to look at the number of shared bits. As you can see, each hash has just around 50% plus or minus 2% similarity with the original hash. When I changed 4 bits at once, it still had about 50% similarity with the original hash. Given that binary data can only come in 1 or 0, it makes sense that two large and random strings of this data would be about 50% similar. They were 100% the opposite of each other, it would be fairly trivial to crack the hash. In fact, through that observation, we know that a good hash should not be biased to giving more or less than 50% similarity because any consistency in that bias would be cryptographically unsound.

Bit flipped	Sha256	Sha 512
H1: This is the original against which all others are compared	6754b1f509d0e77c139aca3c12e8fa7ecd124564c87395b9744c21ed846798e0	af2bdc8845585202a615e4c4bdd90640e18578605ac1e33f8514b1bb3bb930db5fd812ad1b1c098837de7ed8cfeae7130440ffa46a8af8a737f764c7b56ded8b
bits 1, 49, 73 and 113: 256: 134/256 = 52% match 512: 264/512 = 51% match	114fd87f22612264c6d8f4491c18bbcab9bf6b09385c3959956ab99dc5d19121	098833cc088309fae231615905b94b7cfb2e3fc1a49b00311e4a113a4f44230cd53a95fe9e9d2fcb78f86a60e966ea197820df4bd11d67a1805b1f33f128880a
bit 1: 256: 123/256 = 48% match 512: 253/512 = 49% match	f47d526da017634b0ce746dbb643dc48ad0cc838eecca8f64520d1e3c9fa2679	6065de42ab15d8cd7768705aefd5db362bc78fdb23f9b1786def7dea0903497c672c60cf2d24f8de71604e68f0ae26ea1ed5b9171edd3daac7392d9d354e89f1

bit 49: 256: 136/256 = 53% match 512: 266/512 = 52% match	b74ce5761946a5e0ed436fb0b6788a007ade6cbb2dbc179d2c93b389c181f3e4	c1e58e8fc8e922007c0260fc2f123e1493a9b5b615bdb0ffa5b34fd0d711d52ac9abaf3cca7eb811867259585e63080ff50ba46b789a70a8938fa48bf0d071d5
bit 73: 256: 123/256 = 48% match 512: 238/512 = 47% match	633a942f760e3fdf4e263de043c57130d57c94b55c2f3d9671793a8620076c3d	618950fb69ed8fb7516baef2bb8ff70c99dd29cf12221f00f5370a720254c992c3c5b146a97103e10dc8ea2cf0d1432e483625ada8087434a29efb1b483830c4
bit 113: 256: 140/256 = 55% match 512: 246/512 = 48% match	5133126c0ba18f1e1e4e2970eb40d5dbfb8b1757610bb3c9658e431cbd6332ad	d8e811a92e94e8b415996bba3df53d58282f497b322566beb8ceee92cbab4ef7e9ddf822314a7f0d742442b786ae2ffc850d539312f29a781d4e3a122e940652

3.8:

For directions on how to run the script for this question, please see the README.md file in the exercise_3.8 directory attached.

Answers to questions (they are also reported by print statements when the pre-image_resitance.py script is run):

1) Average trials to break weak collision resistance property: 17011420.73
2) Average trials to break strong collision property: 5031.47
3) The strong collision property is easier to break using brute-force.
4) The strong collision property is easier to break because we are looking for any two hashes to match in a single run time, rather than to match to a single target hash as we do while testing the weak collision property. As an abbreviated example we can consider the following:
    a) Strong collision:
        i) Consider we can choose 6 numbers at random from a set of 5 numbers
        ii) In accordance with the pigeon hole problem, it is guaranteed that when the 6th number is chosen, there must be at least one

collision. This means that given an ideal scenario, we've chosen one of each number already, and the next number must be a repeat of one of the previous numbers.

b) Weak Collision:
    i) In this case, we are hoping that one of the many trial hashes we create matches a specific target hash value. Here there are no guarantees
    ii) The same pigeon hole logic **does not** apply. If we were to choose 6 numbers from the same set of 5 numbers, we are **not** guaranteed that there will be a collision.
    iii) Ex. choosing 6 numbers from the set = (1,2,3,4,5). If the number we are hoping to see a duplicate of (our target hash) is 1, we may never see a duplicate value if we continue choosing 5 over and over again, or even if by random we just happen not to choose 1. In this case, while the strong collision property has been broken, the weak collision property has not. We may have duplicates (strong), but we have not reproduced a target (weak).

5) **Implications:** This means that the strong collision property is much easier to break given a brute force strategy, and that if a hash algorithm is secure in this respect, it is also secure in the weak collision property. This is borne by the numbers produced by my python script: Strong Collision ~ 5031.47 and weak collision at 17011420.73.