# Fun with Cryptopals

Tanner Johnson

Developing the Industrial Internet of Things I

Thursday, Feb 3, 2022

## Background

This report is based on my experience solving the cryptopals challenge [1], specifically sets 1 and 2 of these challenges within the context of the Industrial Internet of Things course. These challenges provide a very hands-on experience with implementing, cracking, and understanding some of the key crypto and security technologies that are in widespread use today. As we learned in class, security is the main concern of organizations who are implementing and deploying the industrial internet of things, and for good reason. As industrial processes become increasingly automated and endowed with more and more sensors, the number of vectors of attack for malicious actors are also increased. Moreover, the increased insight these IIoT devices give industrial operators also give nefarious actors increased insight into that industrial process if a device or data stream is compromised. Thus it is important to consider security at all stages of the development process. Moreover it is the job of every individual involved in the development of the IIoT to have a security mindset. In other words, it is not sufficient to hire a sole crypto analysis to solve all your security issues. The power engineer, the RF engineer, the FPGA engineer, all the way up to the data scientist analyzing the data produced by the IIoT must have a security mindset. Finally, in order to achieve something resembling a security mindset one must be at least familiarized with the low level details of these security and crypto technologies and this is where the cryptopals challenges come into play.

## What I did and What I learned

As mentioned above I did the first two sets of the cryptopals challenge. I implemented this in python. Moreover, I made the decision to use no outside libraries beside the random [2] library for generating pseudo random numbers. This made the code as generic as possible and forced me to fully understand all details of implementing the cryptographic algorithms covered by the challenges. This turned out, as one would imagine, to add a bit of time and frustration but was ultimately a wise decision, especially when implementing the full AES-128 algorithm. In the following sections I will simply go challenge by challenge, aggregating a few, and explain what I did and highlight any interesting details of the challenge. Moreover, at the end of each challenge I will detail the key takeaways and what I have learned from this challenge.

### Challenge 1 and 2

Challenge 1 involved creating functions to convert base-64 and hex encoded strings into integer valued byte buffers and turn them back into base-64 and hex string for the purpose of inputting and outputting human readable binary data. Challenge 2 has us create an XOR function that operates on 2 byte buffers, returning a buffer containing their byte wise XOR. The key takeaway of these challenges was really just dusting off my python coding ability. I am fairly comfortable with the language but it has been a hot minute since doing any serious data processing with the language. Also it is worth mentioning that I had not heard of base-64 string encoding so that is something I learned doing these fairly simple challenges.

## Challenge 3 and 4

Challenge 3 was the first challenge where we actually break a cipher. The challenge gives us a hex encoded string and tells us that it was XOR encoded by a single byte repeated. In other words, the key the input string was XORed against is just the same byte repeated the length of the input data. This makes the search space a small 256 possibilities and checking involves simply XORing an input buffer of less than 100 bytes. Thus I chose to simply brute force this search space. However the question remained, how do we know if we broke the code? I solved this by simply ranking each string by how many alpha-numeric ASCII encoded characters the XORed data had. I applied a simple threshold to throw out all but 1 or 2 (depending on the threshold) possible broken ciphers. Then by visual inspection one can determine what the correct cipher is if the output is readable English. Challenge 4 was just an extension of challenge 3. They give us a file of 60 strings, one of which has been one byte XOR encoded. Finding out which one was a matter of applying the code from challenge 3 on each input string and applying the same threshold. This yielded only one possible input string with one possible key.

Looking back on my implementation, this could have been sped up quite a bit. Instead of calculating the XOR over the whole input, if you are checking a possible byte and its xor on the first byte (or any of the bytes for that matter) maps to a character that is obviously not part of an English sentence text (i.e. ASCII 128+ ) then there is no need to continue calculating XORs, that string can be thrown out. The key takeaway here was 1) while XORing input with an unknown key scrambles the data, it is easily recoverable and 2) that brute force is possible given you can craft a small search space and know something about what the structure of the data should look like.

## Challenge 5 and 6

Challenge 5 had us implement repeating key XOR. This is where you have a key of some length and an input of some assumingly longer length. The key is just repeated until it matches the input and then the two pieces of data are XORed producing the cipher text. This was a relatively straightforward piece of code to implement.

Challenge 6 was the first actually challenging challenge. This was for 2 reasons, 1) it was a much longer implementation with many more pitfalls and 2) there was a little theory involved in understanding why the challenge wanted you to break the code the way it asked you too. This challenge had us break the repeating key XOR encryption we implemented in the previous challenge. It had us first implement a bit-wise hamming distance. Then it asked us to find the key length by trying all key sizes in a range. Then for each key size we grab that many bytes from the input, then grab the next key size set of bytes, sequentially. We then calculate the hamming distance between these to chunks, and the key size that minimizes this hamming distance is the correct key size. This confused me at first and required some investigation. It stems from the fact that English language does not produce a uniform distribution of characters and the interested reader can read about the index of coincidence [3], a metric that can be used to prove this fact. My implementation took all such pairwise sequential hamming distance over the entire input and averaged the distance. This proved to be a more stable way to compute the key size.

Once we have the key size, breaking the cipher is just a matter of breaking key size single byte XOR ciphers. So we loop over the input, putting all bytes XORed by the same character in the repeating key together in a buffer. We feed this buffer to our code to break single byte XOR. This produces the key and then we can simply XOR the key, repeated over the input, to the input to get the plain text that was encoded.

The main thing I got out of this challenge was first and foremost a taste for breaking something that at first seemed like it would have a ridiculous search space if brute force was used. In other words a taste for the “orthogonal thinking” mentioned. Moreover, the use of the statistical distribution of characters in an English text to find a plausible key size. Followed by using this key size to bucket the input into buffers all encoded with the same byte. And finally using single byte XOR cipher breaking code we had developed earlier. All this seemed very elegant to me and was definitely a useful exercise. And to conclude one tangible thing I learned that I had not known before was of course about the metric index of coincidence and how it's useful in cryptography.

## Challenges 7-10

Challenges 7 and 10 involved implementing AES-128 in both ECB and CBC mode. These two challenges I ended up doing way more than was necessary but in the end I am glad I did. Instead of calling the openSSL library as suggested, I implement AES-128 by hand in both these modes. In order to do so I used the FIPS 197 specification [4]. We covered the rough details in class, but reading the specification really solidified my understanding of the algorithm and why it has some of the properties that make it useful as a cipher. As covered in class, AES is just repeatedly applying 4 sub-process: add round keys, shift rows, substitution, and mix columns. Adding the round key is rather straightforward and is somewhat obvious as to why it is used to encrypt data. Moreover, this is something we have seen and played with in previous challenges.

The substitution step is very easy to implement, but has some interesting properties. The very first question one would ask is why this substitution box and not some other? Another question one might have is why is it used and how does it affect the actual output? While it is not 100% necessary, I was curios why the AES algorithm the way it is as opposed to some other implementation that would on the surface do the same thing without all the extra steps. I used the book [5] as a reference, specifically chapter 4, which assumes familiarity with modular arithmetic and basic group theory. The substitution box is computed by first taking the inverse of the input byte under the Galois Field of order 256 and then simply doing an affine shift to “obscure the algebraic structure of the Galois Field” [5]. The details of the previous sentence our outside the scope of this report but can be investigate by taking advantage of the references provided. The affects on the output of the S box is simply a highly non-linear function that operates on a single byte that does a sufficient job of adding non-linearity to the output. This is described as adding “confusion” to the output [5]. Noting that the key structure we were taking advantage of in the XOR byte by byte cipher was the linearity of the encryption scheme. This substitution box also has the property that no byte maps to itself and is fast to compute in software using a look up table or in hardware by using the actual operations to compute the inverse in the Galois Field.

The shift rows operation is fairly straight forward, it just shifts the data around. This ensures that, say, only the first byte of the block has any non zero data. Then the shifting operation makes sure the other bytes of the input are influenced and the fact that only byte had non-zero data obfuscated. This is called “diffusion” [5]. The mix rows operation is similar to the shift rows operation in that it adds diffusion to the output by operating on the columns instead of the rows of the data. Its implementation, similar to the S box, relies on the mathematics of a Galois Field and again the interested reader can check out [5] for details.

To conclude, the implementation of AES was somewhat straight forward, but getting a vague idea for why it is the way it is and the mathematical details behind the algorithm took some time. The key thing I learned was simply why the AES algorithm is implemented the way it is, the mathematics of Galois Fields, and some of the key generic properties one looks for in an encryption algorithm i.e. confusion and diffusion.

## Challenge 11

This challenge involved creating an encryption oracle that would prefix and post fix the inputted data by a random amount of random data. However it only appended a maximum of 10 bytes on each end of the input. Also the encryption oracle would encrypt using CBC half the time and ECB the other half, again chosen at random. The objective was to craft an input that would allow you detect what mode it was encrypting in. This was fairly straightforward, one simply inputs enough bytes so that you have to blocks of data that are guaranteed to be your data and not contain of any of the random data appended to the input. Then if those two blocks encrypt to exactly the same cipher text, then its ECB else CBC. The take away from this challenge was really to drive home the point that ECB is stateless and will encrypt the same data to the same cipher text which reveals a ton of information to any malicious actor listening in on the encrypted data conversation.

## Challenge 12

This challenge, while called “Byte-at-a-time ECB decryption (Simple)” was not very simple at least at first glance. I had to do a few iterations on paper before it made sense. The challenge lays out an attack that can crack ECB encryption. You are asked to implement a black box encryption service (something like AES-128 using an unknown random key) that post fixes a chunk of data you want crack. Your job is to figure out what that string is. This is done byte by byte. Let’s assume for simplicity the block size is 4 bytes and the encryption service is e(). The first step to build a dictionary of all cipher texts produced by strings of the form “AAAx” where the last byte i.e. the “x” can be any byte. For all 256 possible bytes we call e(“AAAx”) and store the cipher text block it produces along with the byte that was used. Once we’ve build this dictionary, we call e(“AAA”). Since the block size is 4, the first byte of the secret string will be placed in the last byte of the plain text block. When its encrypted and returned we can simply used the cipher text as an index into our dictionary which will return what the first byte of the secret string is. We can do this for all possible bytes and will eventually reveal the whole string. This challenge is the first that will actually break real world crypto. I had to do a little research to find an example where you send some data to an encryption service and it appends some secret string you’d like to decrypt. Later challenges build on this and make it obvious what types of situations this would useful in. The main takeaway I got from this challenge is simply a practical method of breaking a real, albeit poorly implemented, security system.

## Challenge 13

This challenge can be summarized as the following. Assume we have a profile creation service that takes in an email, encodes it in key value string with the following format: “email=foo@bar.com&uid=7&role=user”. This string is encrypted and stored say in a cookie locally on a users PC. We know it uses AES-128 ECB, the block size, uid will be 1 digit, and unknown key. Using only calls to the profile creation service, find an encrypted text that decrypts and expands to a key value string with admin privileges i.e. contains the key value pair: “user=admin”. This challenge has an obvious application and can be exploited using the fact that ECB is stateless.

We first send an email that contains the word admin padded with null characters such that the admin part of the data is in an encryption block all by itself terminated by null characters.  We then save off the cipher text block it encrypts too. We then craft an email such that the “role=user” is broken up into two blocks such that the “role=” part is in the first block and the “user” part is alone in the second block. Then we encrypt it and replace the encrypted block containing “user” with the encrypted block we computed earlier containing “admin”. Now, the encrypted string will expand to a plain text key value string where the role has value admin. The main skill taken away from this challenge is really one of practical use. One does not need to come up with some obscure use case where this type of attack would be useful. If a user could identify a system using similar way of storing privileges and the attacker could identify these are encrypted with ECB, this attack would basically give you what ever privileges you wanted. Thus the morale is stay away from ECB and maybe don't store privilege vectors in such a transparent manner.

## Challenge 14

This was by far the hardest challenge of sets 1 and 2. In this challenge you again asked to implement an encryption black box. But this time, the data is prefixed with a random amount of random data and is post-fixed with a secret string you want to crack. The goal is to implement a byte at a time hack similar to that of challenge 12 that reveals what the secret string is. Now this took a significant amount of on paper hypothesizing before a solution presented itself. The attack in challenge 12 relied on being able to identify a block boundary such that the last byte of the block contained the byte we where breaking. This allowed us to dictionary all possible 256 byte values and simply index to see what byte is in the secret string. In this challenge, a random amount of data is prefixed to the input data so identifying and crafting an input to put a specific byte of the secret string at a block boundary is difficult but is possible.

The first step is identify the length of the secret string and the maximum amount of random input data prefixed to the input. To figure this out, I simply tried inputs ranging from 0 to 15 bytes and observed the returned number of cipher text blocks over 100 iterations. This revealed a “critical index”. If one inputted strings of length 1-5, the output would be either 10 or 11 cipher text blocks. At 6 bytes the output was always exactly 11 cipher text blocks. For strings of length 7-15, the output was 11 or 12 cipher text blocks. Thus we could conclude the following things. 1) the maximum number of random bytes is 16. 2) the length of the secret string is 138. 3) An input of 6+16n for integer n yields the same number of cipher text blocks regardless of the number of random bytes appended to the input. And finally 4) An input of 7+16n for integer n yields 11+n cipher text blocks when the number of appended bytes is less than the maximum of 16 and yields 12+n cipher text blocks if and only if a maximum of 16 random bytes were appended to the data. Thus we know have a way of detecting when the number of random bytes is 16 and thus can start crafting inputs to put targeted bytes of the secret string at block boundaries just like we did in challenge 12.

Now unlike challenge 12, we start at the end of the string. As in challenge 12 let 4 be e() be the encryption black box, and assume a block length of 4 for simplicity. Also assume our “critical index” is 1 so that inputs of length 2+4n will produce an extra cipher text block if and only if a maximum of block size number of random bytes are appended to the data. So we start by building are dictionary. We create an input string that looks like “xAAA” + “AA”. The first string “xAAA” is the target dictionary index where we loop over x and A is the padding charcter. The the addition two A’s added to the input string are added to the input so we are at that critical length. We then call e(“xAAAAA”) several times until it returns an extra cipher text block. We then index it and loop over all possible x’s to build are dictionary. Finally, we call e() with any string og length 6. When we detect the extra cipher text block we know the last returned cipher text block contains the last character of the secret string along with the padding characters alone in that block. Thus we can use that cipher text block to index into our dictionary and find out what character produced that cipher text, giving the character of the last byte of the string. This is repeated byte by byte to find the full string.

Similar to challenge 12 and 13, this challenge can be easily extended to real world situations. I think the main take away here was to be clever and to think orthogonality. By querying an encryption service when can determine details that might compromise your system. In this case it was some random metric we were calling the critical length. Thus when implementing secure systems, querying your system and seeing if any patterns such as these emerge is probably an important step is securing your devices and services.

## Challenge 15-16

Challenge 15 was just to create a padding stripping function that would be used in 16. Challenge 16 was both easy and enlightening. While learning in class and outside sources. ECB mode was presented as insecure as we have seen in previous challenges. While CBC was presented as fairly secure. In this challenge we break CBC. The challenge involves prefixing and post-fixing your input string with data of known length. The challenge is to create a cipher text containing the string “;admin=true;” with a black box encryption oracle that does not allow you to directly input strings with the key characters “;” and “=”. However, the data prior to the user data may be corrupted, thus we simply input the string “?admin?true?” and save off the encryption. This data happens to lie in the second block and the contents of the first block may be corrupted. So we just bit flip the right bits in the first block so that when XORed with the second block In CBC mode it flips the “?” characters into the correct characters. Thus we now have a cipher text that when decrypted in CBC mode by this black box, will for sure contain the string “;admin=true;” even though the data preceding it was corrupted. So, if a naive privilege checker only looks for this string, you have now given yourself admin privileges.

The take way here is that CBC isn’t fool proof. The fact that each block is the XOR of the previous block gives an attacker to flip bits where he or she desires at the expense of completely scrambling certain blocks of data. Thus, even though AES maybe very hard to break, if an attacker knows a lot about the structure of the data the they are breaking, he or she can take advantage of the underlying algorithm to manipulate the data as they wish. Also the simplicity and ease at which this can was exemplified in this challenge.

## How it Relates to Class Content

In the previous section we detailed what these challenges had us do, what skills and understanding they reinforced, and what the key high level take away was from each challenge i.e. “What I Learned”. In this course, “Developing the Industrial Internet of Things I”, we are given a very quick and hasty overview of security and encryption. These challenges really filled in the details for me. While in class we are given a very course road map with poor resolution, but wide range, these challenges step in and increase the resolution of some of the most important points covered in class. Specifically AES. After watching the course lecture on AES I thought I had a good understanding of it. These challenges made me realize I did not and forced me to do my own and understand it for myself.

While the first two sets really showed a lot about AES, how to crack CBC and ECB, and some very useful attack vectors, it was very much a subset of that presented in class. Some key points covered in class but missing in the first two sets are RSA, Diffie-Helman, asymmetric, etc. However the good thing is the next 6 sets of the challenges cover most topics discussed in class. While I do not have time to do them before submitting this assignment, I hope to finish all 8 sets of challenges as these challenges have really sparked my interest in this field.

## References

[1] https://cryptopals.com/
[2] https://github.com/python/cpython/blob/3.10/Lib/random.py
[3] https://en.wikipedia.org/wiki/Index_of_coincidence
[4] https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf
[5] https://www.amazon.com/Understanding-Cryptography-Textbook-Students-Practitioners/dp/3642446493