Cryptopals 3 Challenges in Cryptography

Tristan Erney

January 26, 2022

Contents

Abstract

First and foremost, I am taking this cryptography internship opportunity despite not having taken the Intro to Cryptology course provided to Computer Science/Security majors here at ESU. Instead, I have decided to take them at the same time. This wasn't a fully intentional choice, but it is one I am glad to have made. It shed light on what I would and wouldn't be able to learn from the course compared to a challenge quest provided by the people behind Cryptopals Crypto Challenges. It was very enlightening and the challenges provided required me to think outside the box and well beyond my comfort zone, something I openly welcomed.

To begin this quest, I decided to program my solutions in C. C is considered to be old and lack in features what other languages like Python, Javascript, or even C++ may have been able to provide. However, since I had not really programmed in C before, I thought this would be a great opportunity to get a feel for the language and what it had to offer. Another decision I made was to not use an abundance of libraries to do this task. I wanted my path to completion of these projects to be a valuable learning experience on how a lot of these algorithms work under the hood. However, for the libraries I did end up using for my solutions I will provide a table toward the end of this report with all the libraries used as well as their uses. With all of this at the start, I set the bar pretty high for myself and the sections following this one are full of my trials, tribulations, successes, and failures.

Set One

For challenge one, we were to convert a hexadecimal string to a base64 string. As I have stated before, I wanted to work out all in the inner workings of the algorithms, so keeping to my word I wrote a header named *conversions.h* to write all of my algorithms which have anything to do with converting from one format to another. As an intermediary between all of the conversions, they get converted to a data type named *byte-string* and then converted to base10, base16 (hexadecimal), or base64. This header would prove to be the base of almost every other solution I created and would be appended to or fixed over time.

Challenge two was a fixed XOR where we XOR two equal length hexadecimal buffers and return the result. For this, we converted the buffers into byte-strings and XOR'd each byte with the byte of the matching index of the other buffer. Doing this gave us our correct result. This solution helped prepare us for Challenge three, which required us to do a single byte XOR against a hexadecimal buffer. Doing this was almost the same as the previous challenge, but instead of two equally long buffers, we XOR a single byte against every byte of the buffer.

Challenge four took what we now know about XORing strings and had us brute force a text file where we were to detect which of the lines had been

encrypted using single byte XOR. This had us test each line and solve for the key that solves the decryption.

Challenges five and six both were to do with repeated XOR. Challenge five had us implementing a repeated XOR on the string:

```
Burning 'em, if you ain't quick and nimble
I go crazy when I hear a cymbal
```

The approach for this was simple enough, but the more difficult of the two was definitely challenge six. In challenge six, we were to perform a breaking of repeated XOR. Creating an implementation and use for hamming distance was simple, but everything after this had me sent into a grinding halt for days. Eventually I was able to figure out how I was to work out the block transposition and was able to rapidly figure out the solution to this challenge.

Finally, we get our first taste of AES in Electronic Cookbook Mode (ECB). This is the first comparison I made from my Intro to Cryptology class I was able to make with these challenges. For this challenge came my first library which I had to import. The name of the library was *openssl* and the header which I had to include from this library was *openssl/aes.h*. The only thing we needed to do was decrypt a base64 encoded file using AES-128-ECB mode decryption using the key "YELLOW SUBMARINE".

The final challenge for set one is challenge 8; detecting AES ECB mode within a file full of hexadecimal encoded ciphertexts. The way we go about finding the solution for this challenge is to prey on AES ECB mode's weakness which is its tendency to create repeating blocks if there is data which repeats itself in the block size. All we need to do it detect which blocks repeat and then record the line where repeats were found. This is a very oversimplified example of the solution but the general idea is there. With that, set one is complete and we now move on to the more challenging problems which can be found in set two.

Set Two

Conclusion

This project was a great experience. Not only was it challenging but brought in great examples of real implementations and attacks that could be seen in real world applications. If I could go back and change anything about my approach to this project it would definitely be which programming language I picked to implement my solutions in. Many people went the smart route of choosing an interpreted programming language such as Python, Javascript, etc. These languages have more high level features which would have been handy when implementing the different solutions. One of the many problems I faced with C was having to deal with segmentation faults from allocated memory not being free'd or allocated memory being double free'd. My implementations worked perfectly, but C has manual memory management so we had to work with handling memory often. One advantage of programming the solutions in C is how portable your code can be. A high level language can add your C code as a feature to their code as to improve performance for instance.

References

Base64: https://www.base64encoder.io/learn/
Base64: https://datatracker.ietf.org/doc/html/rfc4648
PKCS#7: https://datatracker.ietf.org/doc/html/rfc2315