Challenges in Cryptology

Tristan Erney

Abstract

First and foremost, I am taking this cryptography internship opportunity despite not having first taken the Intro to Cryptology course provided to Computer Science/Security majors at East Stroudsburg University. Instead, I have taken them the course and the intership opportunity simultaneously. This wasn't an intentional choice, but it is one I am glad to have made. It shed light on the learning points that would and wouldn't be attained from the course compared to the challenge quest provided by the Cryptopals Crypto Challenges. Within this report, you will find the research and steps I have taken to achieve creating implementations of the challenges provided. Each of the sets completed were given their own sections and can be reviewed in comparison with the research which I have provided.

1 Introduction

Cryptology, according to Britannica, is "science concerned with data communication and storage in secure and usually secret form. It encompasses both cryptography and cryptanalysis"[1]. The usage of concealing data in plain sight has been used throughout history, over time we've been able to use technological advances such as the computer to better secure data over the wire and on storage medium.

For this internship opportunity, I was given the task of completing challenges from the Cryptopals Crypto Challenges. Within this challenge suite, there are eight sets which detail sixty-six challenges which have a wide variety of topics within cryptology. In my implementations, I made it up to the third set, but I have reviewed the challenges and researched throughout all of the sets.

To complete the solutions within this internship I used the C programming language. C is a mature low-level language that was developed in 1972 by Dennis Ritchie at Bell Laboratories. The reason for using this language is because I wanted to gain more experience in C/C++ languages along with low-level programming. Performing these exercises would do that for me. Overall, the structure of this project consists of separate directories for headers, implementations

[1]Britannica Online Encyclopedia definition of Cryptology. https://www.britannica.com/topic/cryptology

of the headers, separate directories for each of the sets which were accomplished in which each challenge was given a separate file, and a Makefile used to compile the project given a file as the parameter. The Makefile was unfortunately not included until much later in the project. Some of the earlier implementations which require external file paths needed alteration to work with the new Makefile compilation method.

Sections that follow this one will be organized by the set name which it describes along with the challenges it contained and a brief description of the research and actions performed in order to complete the set. Section two is Basics, then followed by section three which is Block Crypto, and then completed by section four which describes Block & Stream Crypto. We then finish off with a conclusion to the journey which I took and the resources which were created and used to perform this internship.

2 Basics

The first set is a fairly good spread of introductory challenges. The challenges which are provided include conversions between hexadecimal strings and base-64 strings, operations using XOR, and an implementation of the Advanced Encryption Standard (AES) in Electronic Cookbook Mode (ECB). I implemented most of the components this challenge by myself, so the first thing I went ahead and did was researched the operations for base64 encoding and decoding algorithms from hexadecimal strings.

Challenge one of the set gives us the task of converting between hexadecimal string and base-64 string encodings. I decided on implementing my own conversions for the entire project, so I got started researching into operations for base-64. In order to research for this topic, I found lots of great information from the request for comments (RFC) article 4648[2]. Using this reference I was able to implement a header named *conversions.h* to handle conversions from different string representations. In this case, I created implementations which control conversions between hexadecimal and base-64, however this header will continue to be expanded upon in the future as the scope of conversions broadens.

For challenges two through six, we move onto challenges which consist of XOR operations. The first of which is called Fixed XOR. For the Fixed XOR function to work we need to pass two equal length hexadecimal strings and then XOR the contents of each byte of the strings against each other. For Challenge three, we do the only solution which requires us to complete by hand. Forgoing the instructions, I wrote the solution within the file named *1-Basics/set1_challenge3.c*. The challenge asks us to find the key to a Single-bute XOR encrypted string. The solution requires performing letter frequency analysis as a method of scoring plaintext to evalute which single character was used to encrypt the hexadecimal encoded string. Challenge four is the same as

[2]RFC 4648 - Base-64 specification document. https://datatracker.ietf.org/doc/html/rfc4648

challenge three, but we need to find which line out of a 60 string file has been encrypted using Single-byte XOR. After this, for challenge five we need to implement a Repeating-key XOR function. The internals of this function will take an input character string and the key string where the key string is iterated over cyclically to perform XOR on each byte of the input character string until the end is found. The last of the XOR operations, challenge six, is our first qualifying challenge. The goal is to decrypt a file, find the size of the key using the Hamming Distance[3] function (defined within *headers/hamming_dist.h*), then we can transpose the blocks and solve each block as a Single-byte XOR in order to build a Repeating-key XOR key. This solution is as useful as the challenge says since it is highly revisited in the future.

The last two of the challenges, Challenge seven and eight, both are an introduction to the AES cipher and the different modes which we will be implementing in the future. Challenge seven requires us to use what I believe is the only code within the project which I did not write myself which is the OpenSSL library. As a wrapper around this library, or more in particular the OpenSSL AES library, I wrote functions for all the different modes of encryption and decryption within the *headers/aes.h* file. For this challenge however, we started off with the ECB mode of encryption and decryption. Once this has been passed, we move on to challenge eight, were the goal is to write a function to detect when encryption using AES ECB mode has occurred. To do this, we need to find duplicate blocks of hexadecimal text. If we find any duplicated blocks, then we know that ECB mode has been used since the mode is stateless and deterministic meaning that the same blocks of text will produce the same output. We used the file which was provided in order to determine this.

3 Block Crypto

The second set is as simple or more so than that of set one, but it's clear that set two has lots of roots into the understanding of cryptography. The challenges which are provided within this set are the PKCS#7 padding algorithm, AES in CBC mode, implementation of a ECB/CBC detection oracle, and some exploitations of the algorithms and ciphers we've worked with already. For these challenges, the difficulty wasn't set too high but I still took my time into researching the topics at hand.

For challenge number nine, we were tasked with implementing the PKCS#7 padding algorithm. Within the RFC 2315 article[4] it is showed how the PKCS #7 algorithm works when use. To accomplish this implementation, we create the *headers/pkcs7.h* for padding and de-padding algorithms.

Moving on to challenge ten, we move back to implementing CBC mode for the AES cipher. This AES mode implementation requires us to move away from using the implementation defined

[3]Hamming Distance. https://en.wikipedia.org/wiki/Hamming_distance.
[4]RFC 2315 - PKCS #7. https://datatracker.ietf.org/doc/html/rfc2315

within the OpenSSL AES library and create our own implementation using the AES ECB functions which we created before. The definitions we create will also live inside of the *headers/aes.h* header file for later use. In order to check if the implementation which we create works, we attempt to decrypt the file given to us in the challenge and if successful we should see somewhat intelligible text come from the file. From there in challenge eleven, we can use our newly created AES CBC mode encryption/decryption to create an ECB/CBC detection oracle. The purpose of this oracle is to test whether a given string was encrypted using AES ECB mode or CBC mode. We can do this by using the same detection algorithm we created before to detect ECB mode. The oracle allows us to create different ciphertexts randomly to be ran through the EB detection function. If the detection returns non-zero then ECB mode was detected, otherwise the mode of operation used was CBC mode.

For challenge twelve, we take a step back to AES ECB mode, but now we take a look at creating an exploitation which breaks an encryption without a known key. The process is known as Byte-at-a-time ECB decryption. Using an oracle function we can perform repeated calls to the oracle and perform a pseudo-decryption. The first step was to create a function which serves as our oracle that does AES ECB encryption on the concatenation of a plaintext string and an unknown string. Once we have this, we need to find the block size the cipher uses. To do this, we need to append identical characters to the unknown string until a encryption with a new block is produced. When a new block is produced we know that the block size is the same as the amount of identical characters appended to the unknown string. Now that we know the block size, we can use the unknown string we created to make comparisons which test every possible byte until we find a match between our guess and the actual encryption. We then perform this for all of the bytes until we have successfully decrypted the entire string.

Continuing with ECB mode operations in challenge thirteen, we move onto what is known as ECB Cut-and-Paste. For this challenge, we were required to write a parsing routine to parse a string of text that takes the format of a structured cookie. For this challenge we are to write three functions which are profile_for, get_profile, and create_admin_profile. The profile_for function takes an email to create a profile cookie for with the structure of *email=email@provider.com&uid=10&role=user* and encrypt the string. The second function, the get_profile function, decrypts the string. The real magic happens at the create_admin_profile function, where a new profile is created and then the cookie is padded so the *role=admin* is contained within a new block and can be decrypted and parsed properly.

For challenge fourteen, we need to take a trip to the past back to challenge twelve where we completed the Byte-at-a-time ECB Decryption. The difference between challenge twelve and challenge fourteen is that now we need to create a concatenate a randomized prefix, the attacker controlled string, and the target bytes which we want to decrypt. The goal is the same as before, we want to decrypt the target bytes using the Byte-at-a-Time ECB decryption but with more data which we are concatenating.

Challenge fifteen takes the challenge back to PKCS #7 where now we want to create a function which will validate padding for a block. For the padding of a PKCS #7 pad to be valid, the padding character has to match the amount of times the pad is iterated at the end of the block. An example of this, as they put in the challenge, would be "ICE ICE BABY\0x04\0x04\0x04\0x04". This would be a valid pad since all of the iterations of the pad match the number of instances which the pad appears at the end of the block of text. 0x04 is the hexadecimal value of decimal 4 and it appears in the end of the text four times. Therefore, the pad is true.

Finally, the last challenge of set two, challenge sixteen, consists of performing what is a known attack on the CBC cipher mode which is called a CBC Bitflipping Attack. This attack takes advantage of a known vulnerability with the CBC mode operation which is that a one bit error in a block produces an identical one bit error in the next block of the ciphertext. We use this to our advantage to edit the block of ciphertext to create output we want to insert. In our case, we want to insert the text ";admin=true;". We do this by performing XOR operations on the locations of the injected blocks we wish to alter. The final result after the corrupted block is the text we wanted to inject.

4 Block & Stream Crypto

Set three proved to be a difficult, but not to a degree much higher than the two previous sets before. In this set, we are introduced to a new type of exploitation in the family of pseudo-random number generator (PRNG). The challenge set also tells us that seven of the eight challenges provided within this set are applicable to breaking real world cryptography. In this set we can find topics relating to CBC mode, CTR mode, and the MT19937 PRNG[5]. This set definitely got me deep into research, especially for the MT19937 PRNG since I have never implemented one before.

Our first challenge of the set, challenge seventeen, we create a padding oracle for the CBC cipher mode. According to the challenge, it is possible to create a decryption scheme based on the padding of a block. If we corrupt the ciphertext and it still has valid padding then we can assume the guess we made to corrupt the ciphertext block is a part of the decryption scheme. This challenge is similar to the Byte-at-a-Time decryption we performed on the ECB cipher, however now we have implemented a scheme which works on the CBC cipher mode as well.

The next three challenges, challenge eighteen through twenty, are all about a new mode of operationn known as Counter (CTR) mode. The CTR mode turns AES from a block cipher to a stream cipher which uses a nonce to perform decryption and encryption, which are the exact same process in CTR mode. Challenge eighteen is all about implementing the AES CTR cipher. Using the rules of how a CTR mode cipher operates, we can include the implementation of the

[5]Mersenne Twister MT19937 PRNG. https://en.wikipedia.org/wiki/Mersenne_Twister.

CTR mode in to our *headers/aes.h* header file. From there we move onto challenge nineteen, where we break the CTR mode with a fixed nonce value using substitutions. We can guess the bytes of the keystream by performing some letter frequency analysis on the column of bytes where the keystreams of multiple strings has been encrypted using the same keystream. After we guess the keystream, we can then use it to decrypt all of the encrypted strings and get a somewhat human readable string of text. I say somewhat because some of the characters didn't have full compatibility with he keystream, but they are still able to be deciphered. Lastly, challenge twenty, we were to break the CTR mode statistically, but looking back to my solution from challenge nineteen I realized I had already solved my solution this way, so I decided to forgo this challenge and start with the challenges for the MT19937 PRNG.

For the last section of this set, challenges twenty-one through twenty-four, we complete challenges for the MT19937 PRNG. I was not able to complete all of these in time of the submission, but I did complete up to challenge twenty-three so I will discuss about those challenges. Challenge twenty-one, while simple enough did take some time to fully understand and perfect. The goal of this challenge is to implement the MT19937 PRNG. There is a great section within the wikipedia article on MT19937 which gives the pseudo-code for how the code should work. This was a great resource and allowed me to understand the different sections of the number generator which helps in a later challenge. After the implementation is complete, we can then move onto challenge twenty-two which wants us to crack the seed of a MT19937 PRNG. The seed is vital because it is the root for generating the random number sequence. If this seed is broken, then the rest of the numbers in the sequence can be calculated and predicted. We can do this one of two ways, simulated (we simulate the passing of time), or real-time (we perform the crack as intended and wait the given amount of time). To do this, I created a function called crack_mt19937 which performs the passing of time for all of the numbers of seconds between 40 seconds and 1000 seconds. To make the simulation work, we needed to provide some time deltas which would simulate the passing of time. To do this in real-time, we could exchange the code which provides the time deltas and just perform with the passing of time. Finally, we move onto the last challenge which was performed, challenge twenty-three. Challenge twenty-three requires us to clone the MT19937 PRNG from its output. To do this I wrote two functions which I then included into the *headers/mt19937.h* header file. The two functions are mt_untemper and mt_clone_state where the mt_untemper is an exact inverse of the the tempering function[6] within a MT19937 PRNG (defined as the function mt_temper) and the mt_clone_state allows us to clone the PRNG itself.

[6]Mersenne Twister untempering function. https://crypto.interactive-maths.com/frequency-analysis-breaking-the-code.html.

5 Conclusion

Not far into researching the various sections of this challenge, I learned programming the solutions using C was not a popular take. Many of the dynamic scripting languages such as Javascript, Python, PHP, and many others were preferred as they took away the need to create the abstractions necessary to completing the challenges. Instead of getting discouraged and halting progress, this became even more of an incentive to solving the complex problems through research and analysis rather than copying the solution from an outside source.

In terms of time, the process that took the most amount of time to complete was the research itself. Understanding the nuances for multitudes of algorithms specified was time consuming but could be considered the most important part of the entire process. If not for understanding the algorithms or exploitations being performed then the solutions which I created wouldn't have been written at all. I wasn't able to complete all of the challenges as I intended to, but this will become the groundwork for completing the entire suite of challenges and documenting my path towards completion.

Given the chance to start again, I would have began with implementing a plan to limit the amount of memory allocations which were introduced in the solutions' headers and implementations. Using the amount of resources I did caused a lot of complications during run-time where the program would introduce a segmentation fault, creating lots of time which required debugging using gdb. In light of this however, these errors in my programming made me more conscience of the dangers introduced while working with memory and allowed me to change my style of programming in the latter half of the project to not use as many allocations. In the end, all of the takeaways are positive and I hope to keep completing these challenges and ones like them in the future.

Appendix

Headers Glossary

Header	Usage and Info
aes.h	Defines AES encryptions, decryptions, detections, and exploitations for ECB, CBC, and CTR modes
assert.h	Defines macros for run time assertions. *assert(cond)* prints a default error condition then exits the program. *assertm(cond, msg)* prints a user created message when a condition is not met then exits the program
conversions.h	Conversions between strings, hexidecimal strings, and base64 strings
hamming_dist.h	Implementation of Hamming Distance which is used for repeated XOR attacks
letter_freq.h	Perform letter frequency analysis on a *byte_string* to produce a score for that given data
memory_output.h	Outputs the data stored within a *byte_string* in the form of hexidecimal encoding on the left side, and text output on the right side
mod.h	Perform modulus operations on positive and negative numbers since the builtin modulus operator in C only produces results for positive numbers
mt19937.h	Implementation for the mt19937 32bit pseudo-random number generator
pair.h	Data structure for creating and storing a tuple pair
pkcs7.h	PKCS #7 algorithms for padding bytes to fit into a specified block size. Used for AES encryption and decryption
xor.h	Perform different forms of xor functions such as single, repeated, hexadecimal, or over a *byte_string*

Libraries Glossary

Library	Usage and Info
OpenSSL	Contains the implementation for AES which we use to define our AES implementations

References

[1] Simmons, Gustavus J.. ”cryptology”. *Encyclopedia Britannica*, https://www.britannica.com/topic/cryptology.

[2] S. Josefsson. ”The Base16, Base32, and Base64 Data Encodings”. *Internet Request for Comments*, 2006, https://datatracker.ietf.org/doc/html/rfc4648.

[3] Wikipedia Contributors. ”Hamming distance”. *Wikipedia, The Free Encyclopedia*. 25 March 2022, https://en.wikipedia.org/wiki/Hamming_distance.

[4] B. Kaliski. ”PKCS #7: Cryptographic Message Syntax Version 1.5”. *Internet Request for Comments*, 1998, https://datatracker.ietf.org/doc/html/rfc2315.

[5] Morris Dworkin. ”Recommendations for Block Cipher Modes of Operation”. *National Institute of Standards and Technology*. 2001, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[6] Wikipedia Contributors. ”Mersenne Twister”. *Wikipedia, The Free Encyclopedia*, 26 April 2022, https://en.wikipedia.org/wiki/Mersenne_Twister.

[7] ”The Mersenne Twister”. https://www.maths.tcd.ie/ fionn/misc/mt/

[8] Daniel Rodriguez-Clark. ”Frequency Analysis: Breaking the Code”. 2013, https://crypto.interactive-maths.com/frequency-analysis-breaking-the-code.html.