Roll Your Own Crypto

Presenter: Joshua Stover, CIS Department · Mentor: Dr. Sarbani Banerjee, CIS Department

Abstract

With data breaches on the rise and the looming threat of vulnerabilities in established encryption algorithms inching closer, data security is a growing need. Security must be considered at every step in a piece of software's lifespan, and in every component therein. This research project will employ a novel message-digest algorithm that will generate 512-bit, 256-bit, or 128-bit keys from a user selected passphrase. The message-digest algorithm will use a call to the system time to salt the hashes it generates, ensuring the uniqueness of keys even if passphrases are similar. In addition to the message-digest algorithm, it will implement an unique symmetric-key encryption algorithm using the aforementioned key to encrypt and decrypt files. The software will be written in C++ and compiled for use in Windows 10 and Ubuntu Linux. A simple interface for each platform will allow file selection and choice of encryption level. The project will be created using Visual Studio Code in conjunction with Qt Creator. The presentation will show machines running the software, encrypting and decrypting data from local and removable disks. The poster will examine the mathematics and the code used by the algorithms, as well as an exploration of use cases. The project aims to show that creating novel encryption algorithms can increase the data security of an organization or an individual without being any more cumbersome than implementing an existing algorithm.

Inspiration

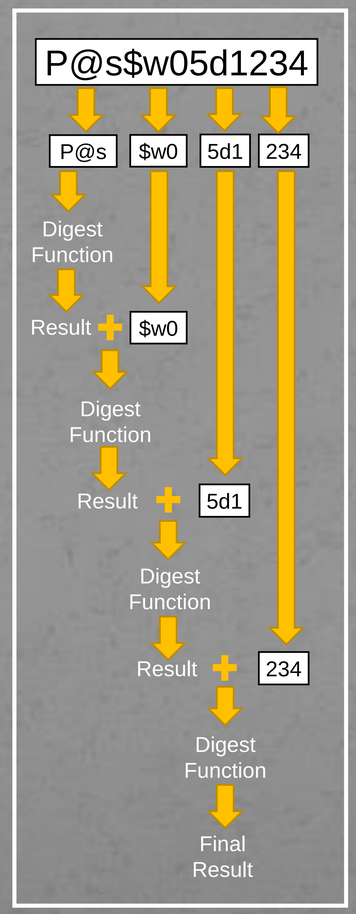
In the seminar on Python last semester with Dr. Banerjee, I had recreated the MD5 and SHA-2 message digest algorithms without any libraries, and felt that taking a step further into cryptography would be fitting. Though working with Python was interesting, I wanted to create something that I could compile and run on any system without the installation of any additional software.

Encryption and Hashing

The encryption used in this program follows a very simple pattern. After the user's key is generated by the hashing algorithm, it can then be used to encrypt files. This same key is needed to then decrypt the files. To encrypt, the program first appends the key to the end of the message, ensuring that the message is longer than the key. Then the program, starting at the first byte of available data, will perform an XOR between the key and the matching sized chunk of the message. After this XOR is performed, the program shifts the message one byte to the left and repeats until the end of the message is reached. To decrypt a file, the reverse is done. Starting at the end of the file, the key and file chunk go through an XOR, then the message is shifted one byte to the right.

The hashing algorithm is more complex than the encryption, but the diagram below shows the general pattern. After a user has entered their password, the string is converted to binary data for manipulation. Once this has been done, the system appends the current system time to the string. Depending on the length of the string after this, zeroes may need to be appended to ensure the digest algorithm has equally sized blocks as input. The padded string is then broken into these blocks, and the first is fed to the digest function.

The digest function takes the password chunk and performs some operations on it using a set of constants (in this case, the constants are sequences found in the decimal portion of pi). The operations that are performed are based on the numerical value of the chunk, meaning that a larger number may have a totally different set of operations than a smaller number. In the example to the right, the result of the digest function when given the first chunk is then added to the next chunk, and that result is given to the digest algorithm. In this program, the chunks are operated independently until the final step, where they are composed into a hexadecimal string that will then be the user's key.



Method

While working on this project a great deal of time was spent designing the message digest algorithm. The goal was to make something that would run in a reasonable amount of time, but still generate usable results. To use a message digest algorithm to generate encryption keys, the output must be unique for any given data set and impossible to reverse into the original text. In addition to designing an algorithm that should do these things, I've implemented a mechanism to "salt" the hashes, meaning that an additional number is added at some point in the process to make sure that similar or identical data sets do not produce the same keys. This is done with a call to the system time, which is used to rotate parts of the user's password as it is converted to integer values. After this, the string is buffered with zeroes to a workable length. Though I'm reasonably sure that collisions do exist in my algorithm, the amount of time and processing power required to test this are beyond my means. The encryption and decryption are a simple XOR through the file that the user wants to encrypt or decrypt.

Findings

The digest algorithm can create three differently sized keys from a password, and the salting method works as expected. The interface was built entirely in Qt because of time constraints, but is still able to be compiled for Windows and Linux. Source code and executable files, once completed, can be located at https://github.com/JoshuaStover/roll_your_own_crypto

Conclusions

Though the initial assumption was that building a novel encryption and hashing algorithm would be no more troublesome than implementing an established set of algorithms based on their documentation, this was not the case. It appears that the task is achievable for the average developer, though far more time consuming than using existing code. It is also worth noting that proper testing of these algorithms is time consuming and heavy on compute resources, so the ability to verify that collisions won't occur is beyond the scope of this research.