Titan Newman

Security Algorithms

Professor Kippins

5/15/2022

AES File Encryptor

Abstract

For an encryption project, I was tasked with creating an encryption application. I chose to create a Windows executable that takes a variety of different user inputs and proceeds to either encrypt or decrypt a file based on the user's choices/input. The file can be encrypted via two different AES encryption variations, where both follow the base AES encryption cycle, but add their own functions and procedures. The application does not save user inputs, so users are required to remember their choices for encryption variations and passwords. This allows the program to remain the 'middleman' and does not require passwords to be saved securely in the application. The main encryption variation for this project is GCM (i.e. Galois Counter Mode) which is an extension of the CTR variation and uses a stream cipher to encrypt and decrypt the data. GCM is the top choice of AES encryption (i.e. based on NIST and can use 256-bit cipher key). The CBC (i.e. Cipher Block Chaining) variation is also included, but was the preferred variation based on the book "Cryptography Engineering: Design Principles and Practical Applications", however it is a little dated.

Introduction

My task for this project was to produce a project that works on either a Desktop, an Android device, or an iOS device. This program/project must utilize a security encryption method and allow for user interactions with it. I decided that my project will be a Microsoft Windows executable application which will take a file in as input and encrypt it using AES (with 128 bits). The application will not implement authentication (i.e. external to encryption authentication) as the files will not be leaving the local computer, it creates a system key (i.e. a symmetric key) based off a user password, and it will enable the file to be decrypted based on a key given by the user. This means that the application will take in a specific file, apply the AES encryption standard, and save it. We can then choose to either decrypt the file (i.e. based on its key) or leave it as is. There are already applications around that do this, however this application strictly uses Java's AES libraries such as 'crypto' (i.e. a Java's security and encryption library), as these libraries follow the encryption standards of NIST which also allows for "cleaner/maintainable" code. The rest of this paper dives deeper into my project, and how I have approached this task. The paper will touch on some related work, the methodology behind the creation of the project, the analysis of the workings, the application's testing, and a brief conclusion.

Related Work

There are a lot of encryption programs available and many of them have very similar traits. Many of these applications not only cost money, but also enable cloud back-up functionality. Some applications that I based my project off include: NordLocker; Steganos; AxCrypt; CryptoForge and Cypherix SecureIT. Most of these applications use AES-256 and 4096-bit RSA for encryption and authentication, as some of their files are stored in the cloud and a secure connection needs to be established (e.g. NordLocker also uses ECC *[Elliptic-curve cryptography]* for both assigning public and secret keys). All these applications have their own pros and cons, however the commonality is the fact they save the file's encryption password/passphrases. This means that anyone who can gain access to the application can also gain access to the encrypted file's keys. This was a problem that I didn't want for my application, which led me to only allowing the application to take in a password once while running. This means that the passwords aren't saved and keeps the password's safely out of the application's responsibilities. I am unsure of the languages used by the related software, however most of them use widely accepted encryption libraries which helps to keep them secure. These libraries allow a plethora of tests to be conducted (i.e. tests for encryption errors), but also allows these encryption standards to become widely accessible. As these libraries are used frequently, I decided to incorporate them into my project to allow for the latest and strongest versions of AES encryption.

Methodology

For this project, I decided to work from the ground up, but stuck to using Java as the language as I could fall back onto my AES encryption function if the libraries didn't quite work out. Java has an extensive import library which allows for faster and cleaner code to be produced. I started off by creating the UI for the application, where ease of navigation for the user was the focus. I stuck to using Java's *'javax.swing'* import which holds functionality for both free user selection (i.e. selecting files) and for user inputs (i.e. input passwords, selecting what encryption variation is to be used, and whether the file would be encrypted or decrypted). I then made sure that I was able to properly read a file's data and save it locally. The next stage was to run through the encryption/decryption processes for the data in a file. Based on the user's input, the application would call the required functions to incorporate the necessary AES variation. For this, I used the imports "*javax.crypto*" and *"java.security.spec"* to successfully encrypt/decrypt files (i.e. the data inside of the files). The actual encryption/decryption functions create ciphers based off the imports, which depending on the variation, either use a block cipher or a stream cipher encrypt/decrypt the data - I will dive into the differences as how this work in the analysis section. The awesome outcome of using Java's imports is how incorrect secret keys (i.e. basically user's passwords) can cause a cipher to halt when the first byte can be encrypted correctly. This allows the user's encrypted file to stay encrypted and does not require an extra decryption cycle after a failed decryption. The next stage was to make sure both types of the encryption variations would work and wouldn't allow cross encryption or decryption. At the end of the project, I tested both AES variations, which I will talk about in the upcoming testing section.

Testing

To test that everything was working, I first started with a base CLI, which allowed me to pass in files and test both encryption and decryption. As testing and allowing the encryption-decryption process to completely run through was a necessity, I tested manually before the front end (i.e. the Java 'UI') was implemented. Base cases including creating files and allowing the encryption process to complete without error for both AES variations. The next step was to make sure the decryption process was able to complete successfully (i.e. based on the correct password), which was successful. The final step was to test encrypting and decrypting with different passwords, which ended in an exception to be passed the cipher (i.e. exactly what we wanted). Another important test was to ensure if different encryption variations were used for encrypting and decrypting, it would lead to either and error or an exception. This is important as using the same variation was needed for a file to go through the entire process. When CBC was used as the encryption variation and GCM was used as the decryption (and vice versa) the file would fail to decrypt (i.e. which is what needed to happen). There are now 3 automated test cases for this process, which can be found here: *'mscs630newman\prj\writeup\code\demo\src\test\java\com\file_encryption\testCaseFiles'*.

Analysis

This application/project uses 2 different AES encryptions, where the first is CBC and the second is GCM. They both incorporate AES's encryption cycle (i.e. add round key, substitute bytes, shift rows, and mix columns), but the main difference between them is that CBC is a block cipher (i.e. old and not the standard), and GCM uses stream ciphering. The functions implemented in conjunction with both variations use of Java's *'javax.crypto'*, which holds the actual AES encryption/decryption functions/ciphers. The next two paragraphs describe their perspective variation and how it ties into this application.

Firstly, we will look at CBC. As CBC is not a stream cipher, it needs a padding function to allow the 128bit encryption to work properly. This means when the data that we are encrypting isn't 128 bits, we use PKCS 5 as the padding method (i.e. fills the remaining data to 128 bits). PKCS 5 is just a padder for the 128-bit key, which pads the key based off using the modulo to derive the needed bits. We see that this schema (i.e. a pseudorandom function, ex: hash-based message authentication code *[HMAC]*) is used in both AES and RSA to reduce brute-force attacks and create a secure cryptographic key. CBC uses both the IV (i.e. the initialization vector) and XORs the plaintext together (i.e. for this application we converted directly to bytes to allow the cipher/algorithm to work more efficiently), which gives the first round of the AES encrypted data. The algorithm then moves to the next block and uses the encryption data (i.e. from previous) to XOR with the next block of bytes and continues until it reaches the end of the data or the padded data. This variation is the step above the ECB variation, as it is randomizing the plaintext (i.e. bytes) used in the previous block, which left ECB with a vulnerability. Sadly, CBC cannot be parallelized, which was one of the reasons why it was slow and not the AES standard in the eyes of NIST.

The last variation is the GMC, which is at 256 bits (i.e. but processes at each 128 bit), and the actual key size's relation is 256 bits. The generalized encryption works with a plain text and a secret key which, when used together in the cipher, give you a corresponding cipher text. Before explaining the GCM variation, I need to explain its precursor, CTR. CTR first encrypts the IV and uses it as a counter, where after each round it encrypts the counter to help add block noise. It is also a stream cipher which encrypts each binary bit in the stream. CTR does need a nonce, which is used to create the actual key stream. The textbook (as it’s unfortunately a little old) stated that CTR was the best choice, however GCM is the newest in the line of AES encryptions to be the top choice. GCM is an extension of CTR and uses a counter for each block it encrypts, where after each round, the cipher gets the new block number which is first combined with the IV and encrypted with AES (i.e. the block cipher), where the result is then XOR'ed with the plaintext to produce the ciphertext. This allows block ciphers to be turned into stream ciphers. Java's crypto cipher allows this variation to run in parallel for both encryption and decryption, as each block can be encrypted by itself. Another notable factor of GCM is it contains an authentication tag. As this is a 128-encryption standard, it uses a key length of 8 32-bit integers and goes through 14 rounds. These rounds are like CTR, where the last round only computes the substitution, shift rows and adds the round key with counter. The authentication tag is computed by using the counter and hash function on the first block which then goes through the AES encryption process, and ties into the first and second AES rollovers (i.e. where you go through each 256-bit encryption processes, and each time the counter is also increased). This allows us to authenticate each cipher text (i.e. 256-bit portion) to make sure it was computed correctly. The main function used in this variation is the GHASH function, which computes the authentication tag: GHASH(H, A, C) = X m+n+1, Where: H = E k(0^128) [block cipher/AES], A: the data that is not encrypted (i.e. not encrypted and authenticated), and C: is the cipher text of size 128 blocks.

Conclusion

In conclusion, both AES variations are strong, but as time has passed and technology has moved forward, CBC is no longer the standard based off the textbook, and GCM has come to become to standard as most stream ciphers have. This application allows single file's to be encrypted or decrypted based off both a password and an AES variation chosen by the user. Without user passwords being saved or stored, the liability of remembering the password falls onto the user, hence off the application. The application also warns the user when using an incorrect password and doesn't delete the said file.

Bibliography

- https://en.wikipedia.org/wiki/PBKDF2
- https://en.wikipedia.org/wiki/Galois/Counter_Mode
- https://iopscience.iop.org/article/10.1088/1742-6596/1019/1/012008/pdf

- N. Ferguson, et.al.,Cryptography Engineering: Design Principles and Practical Applications, John Wiley & Sons, 2011
- https://docs.oracle.com/javase/7/docs/api/java/security/spec/KeySpec.html
- https://docs.oracle.com/javase/7/docs/api/javax/crypto/SecretKeyFactory.html
- https://docs.oracle.com/javase/7/docs/api/javax/crypto/KeyGenerator.html
- https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html