Secure Cloud File Storage System

6 April 2020

John David Ford

I. Table of Contents

II. Executive Summary

The goal of the "Secure Cloud File Storage System" project is to securely encrypt and transport text files from a client system to a local server utilizing hybrid encryption. The specific encryption algorithms utilized were AES128 with a user defined choice of 16, 24 or 32-bit keys, SHA-512/256 and RSA. The project was built in the Python programming language and the data server was hosted on an Ubuntu virtual machine.

Successfully using the system requires the following events:

1. User selects the text file to be encrypted.
2. The system splits the given text file into 3 separate fragments. Each fragment containing 1/3 of the original text files contents.
3. The system applies a unique encryption algorithm to each fragment in round robin fashion. Fragment 1 receives AES128, Fragment 2 receives RSA and Fragment 3 receives SHA-512/256.
4. All fragments are sent to the local data server and stored.
5. All fragments are sent to the client pc and ready to be decrypted.
6. The system decrypts each fragment and reconstructs the original text file to be displayed to the user.

Data analytics were performed on the system with varying text file lengths to determine system effectiveness. Having utilized pre-built encryption Python libraries resulted in efficient encryption which allows further exploration of the systems use outside of a lab environment.

III. System Description

i. Needs Assessment

Privacy, security and on-demand file access don't generally coincide together. Specifically, users who don't care about securing their personal information aren't willing to take the time to secure it unless they have been negatively affected. This system attempts to uniquely and efficiently secure an impatient user's data without impeding their general computer use.

ii. Design Constraints and Standards

The language implementation requirements were PyCharm Python 3.7 with sys, time, socket, pathlib and Crypto.* libraries. The operating system requirements were Windows 10, VMWare vSphere Client and Ubuntu 19.10. The system has no economical constraints due to the server being owned, not rented.

iii. Security and Privacy Considerations

The system is security centric in its design, especially when transferring file fragments from a client pc to the data server through encrypting the raw data before file transfer takes place. During the encryption process, encryption keys are locally stored on the client pc as obscure file types to prevent easy access. Privacy is considered through the storage of received encrypted data and through the deletion of any file trace on the server after a client requests their previously sent files return to their pc.

iv. System Design Diagram

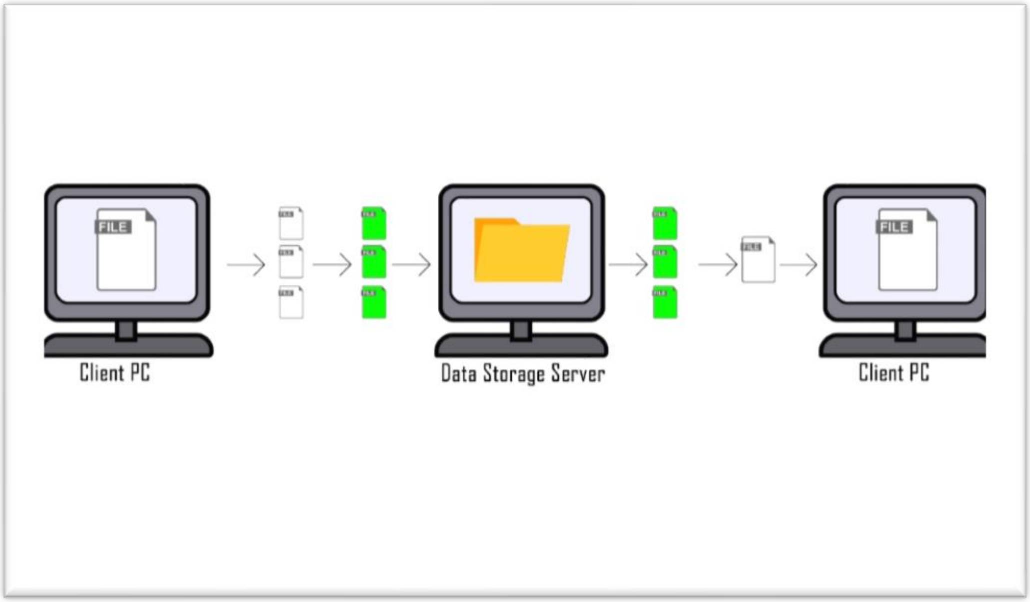


Figure 1 - System Diagram

IV. Detailed Implementation

i. Hardware Requirements

i. Client-side development was performed on a Windows 10 Surface Pro 3.

ii. Server-side development was performed on a 128GB Ubuntu 19.10 Virtual Machine housed through vSphere Client hosted on an HP ProLiant MicroServer.

ii. Software Implementation

i. File Selection and General Program Execution

When the program starts, a user is asked to enter a file name and is then presented with a 7 choice menu asking if the user would like to: (1) Split File, (2) Encrypt File, (3) Decrypt File, (4) Change File, (5) Send Files, (6) Get Files or (7) Exit. The program utilizes the pathlib library to check if a directory exists for the entered file and if one doesn't exist a directory is created and labelled the files name. If an unrecognized file name or an non-existing file name is entered when a user attempts to select any option other than "Exit" they will be prompted with a file not found error and asked to enter a valid file name.

ii. File Splitting

The program takes the given filename and checks if the file exists as previously stated. When the file is found, the program reads the given text file and stores the contents in a TextList as words separated by the space character. To create the 3 split files, the program takes the length of the TextList, divides by 3 and then feeds 3 separate lists with the contents of the TextList from 0 to a, a to a+a and a+a to the end of the TextList. A new file is created for each list labelled GivenFileName1, GivenFileName2, and GivenFileName3. Each new file is then filled with the contents of one of the newly created lists.

iii. File Encryption

The encryption process begins by checking if the file exists and if it does it creates a file for the AES nonce labelled GivenFileNameNonce.txt and encrypts in the following order: AES, RSA with AES, SHA-512/256. GivenFileName1 will receive AES, GivenFileName2 will receive RSA with AES and GivenFileName3 will receive SHA-512/256.

The AES encryption is performed using the Crypto.Cipher library. The function reads GivenFileName1.txt's contents, prompts the user to enter a custom 16-bit, 24-bit or 32-bit key used to generate a cipher and nonce. The nonce is copied into the GivenFileNameNonce.txt for later retrieval. The AES cipher is then applied to the contents of the text taken from GivenFileName1.txt. Finally, the encrypted contents replace the original contents of GivenFileName1.txt.

The RSA encryption utilizes the previous AES-128 encryption and the pathlib, Crypto.PublicKey and Crypto.Random libraries. The function reads GivenFileName2.txt's contents, generates 2 new CMS (S/MIME) files labelled "GivenFileNamePriv.pem" and "GivenFileNamePub.pem", generates an RSA public private key pair and loads the CMS (S/MIME) files with their respective keys. An RSA cipher is created using the previously generated private key. Then, a random 16-bit session key is generated and encrypted via the RSA cipher. The encrypted session key is used as the key for the AES encryption that encrypts the contents of GivenFileName2.txt.

The SHA-512/256 encryption utilizes the pathlib and Crypto.Hash libraries. The function reads GivenFileName3.txt's contents, creates a new file labelled "GivenFileNameO512C.txt" (meaning Original 512 Contents) and copies the original raw text data into "GivenFileNameO512C.txt" for later reference. The contents are then sent through a SHA512 hash that is truncated to 256 bits to avoid length attacks and output to GivenFileName3.txt.

iv. File Transfer and Storage

The file transfer process begins based on whether the user elects to send their files to the server or get their files from the server. When the files are transferred, they are transferred one by one. The function utilizes the socket and pathlib libraries.

i. Send Files

If the user wishes to send their files to the server, the program reads the contents of GivenFileName1.txt, creates a socket directed towards the ip of the data server on port 8888 and connects the socket to the server. On the client side, the first message sent to the server is the user's option of store, the second message sent is the size of the file contents and the third message is the data itself. Once the data is sent, GivenFileName1.txt is deleted. This process is repeated for GivenFileName2.txt and GivenFileName3.txt.

On the server side when the first message is received it sends a "Buffer size set." message, when the second message is received it sends a "Data Successfully Sent" message and when the final message is received a while loop is opened with a buffer to catch all of packets containing the data until the buffer size is less than the size of the caught data. The server checks to see if a directory already exists for the files that were sent and if one doesn't, a directory labelled "GivenFileName" is created where all 3 sent files are stored.

ii. Receive Files

If the user wishes to receive their files from the server, the send process is reversed and repeated. On the client side, the first message sends an empty message to bypass the send’s buffer size setting and receives a “Buffer size set” message, the second message indicates the option of get and receives a message of the file size, the third message is the same as the second message. However, the third message receives the data from the server and a while loop is opened with a buffer to catch all of packets containing the data until the buffer size is less than the size of the caught data. When all of the data is received, GivenFileName1-GivenFileName3.txt files are created and filled with the received contents.

On the server side, when the second message is received, it reads the contents of GivenFileName1.txt, takes the length of the contents and sends the length to the client. When the third message is received, the server reads the contents of GivenFileName1 and sends the data to the client. Once all of the data is sent, GivenFileName1.txt is deleted. The process is repeated for GivenFileName2.txt and GivenFileName3.txt. Once all 3 files have been deleted, the GivenFileName directory is deleted.

v. File Decryption and Presentation

The file decryption process begins by checking if the file exists and if it does the AES decryption begins.

The AES decryption utilizes the Crypto.Cipher library and begins by reading the GivenFileNameNonce.txt file and prompting the user to re-enter the custom 16-bit, 24-bit or 32-bit key used for encryption. The AES cipher is re-generated with the given key and previously stored nonce. The cipher is then used to decrypt the read-in contents of GivenFileName1.txt. Once completed, the decrypted contents replace the encrypted contents of GivenFileName1.txt. Finally, the GivenFileNameNonce.txt is deleted.

The RSA decryption utilizes the previous AES-128 decryption and the pathlib, Crypto.PublicKey and Crypto.Random libraries. To begin, the program reads the previously stored RSA private public key pair files GivenFileNamePriv.pem and GivenFileNamePub.pem and uses the private key to re-generate the RSA cipher. The program then reads in the encrypted contents of GivenFileName2.txt. The re-generated cipher is then used to decrypt the previously encrypted session key to use with the AES decryption. The GivenFileName2.txt contents are then decrypted, and the decrypted contents replace the encrypted contents. Finally, the GivenFileNamePriv.pem and GivenFileNamePub.pem files are deleted.

The SHA-512/256 decryption utilizes the pathlib and Crypto.Hash libraries. To begin, the GivenFileNameO512C.txt and the GivenFileName3.txt files are read in for comparison. Before comparison, the

GivenFileNameO512C.txt is hashed exactly how GivenFileName3.txt was. The hashed contents of both files are compared to each other, if they are the same, the program writes the original contents of GivenFileName3.txt over the encrypted contents of GivenFileName3.txt and the GivenFileNameO512C.txt is deleted. If the hashed contents are not the same, the user is presented with and error detailing the contents are not similar indicating file tampering.

Once all 3 files have been decrypted, the contents of each file are read into a list, a file labelled GivenFileName.txt is created, the new file is filled with the contents of the list and the GivenFileName1-GivenFileName3.txt files are deleted. This final process leaves the user with a directory labelled GivenFileName containing the GivenFileName document.

V. System Test and Results

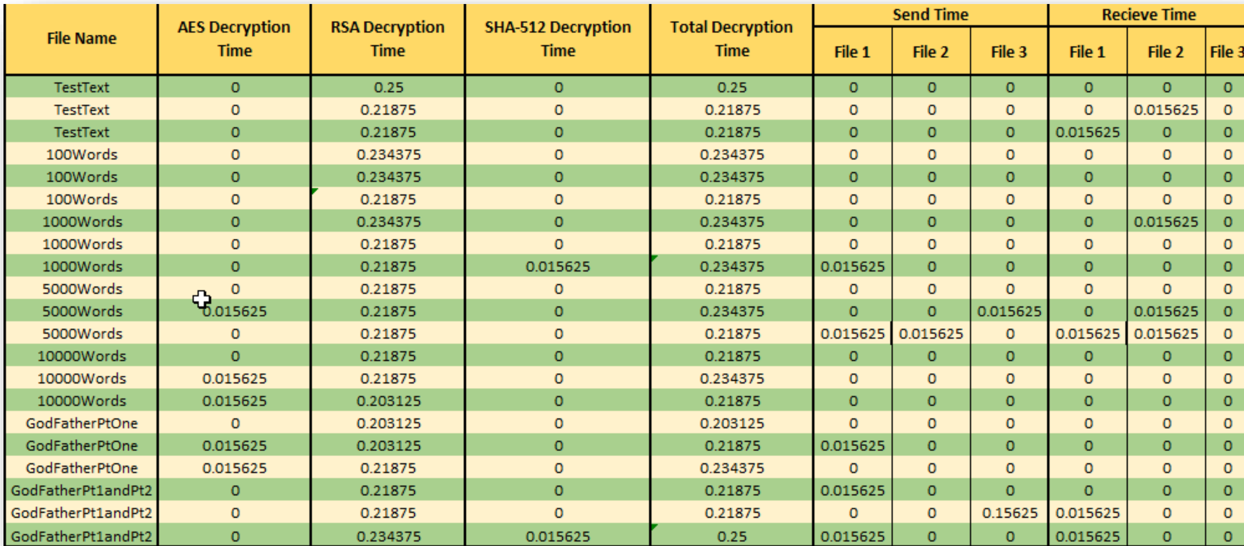
The system was tested by selecting, encrypting, transferring and decrypting 7 different files with lengths ranging from 25 up to 65,158. Each file was tested 3 separate times with the only variation being the AES encryption key. The first test was with an AES key of 16 (“qwertyuiopasdfgh”), the second test was with an AES key of 24 (“qwertyuiopasdfghjklzxcvb”) and a final test was with an AES key of 32 (“qwertyuiopasdfghqwertyuiopasdfgh”).

Figures 2 and 3, located on pages 12 & 13, show the encryption, decryption and file transfer timing results, displayed in seconds, from the various testing that was performed on the system. Analysis of the encryption results indicates the only consistent timing hurdle is the RSA encryption. Further, the overall inconsistency and lack of pattern in the “RSA Encryption Time” indicates the timing is more dependent on the random number that was generated, encrypted and then used as the key to the AES portion of the RSA encryption than the file length itself. Analysis of the decryption and file transfer results indicates there are no significant timing hang ups slowing the system down. However, the low timing results of the file transfer may be due to the server being hosted on a local network the client was connected to, therefor significant packet travel was not required.

Overall, the system may be viable outside of the lab environment with further research and tweaking required for the RSA portion of the encryption process.

File Name	File Length (Words)	AES Key Size (16, 24 or 32)	AES Encryption Time	RSA Encryption Time	SHA-512 Encryption Time	Total Encryption Time
TestText	25	16	0.015625	3.90625	0	3.921875
TestText	25	24	0.015625	4.421875	0	4.4375
TestText	25	32	0	6.78125	0	6.78125
100Words	100	16	0	12.125	0	12.125
100Words	100	24	0	1.359375	0	1.359375
100Words	100	32	0	7.328125	0.015625	7.34375
1000Words	1195	16	0	5.1875	0	5.1875
1000Words	1195	24	0	14.65625	0	14.65625
1000Words	1195	32	0	4.453125	0	4.453125
5000Words	5000	16	0	6.796875	0	6.796875
5000Words	5000	24	0	11.828125	0	11.828125
5000Words	5000	32	0	2.859375	0	2.859375
10000Words	10000	16	0	2.875	0	2.875
10000Words	10000	24	0	3.15625	0	3.15625
10000Words	10000	32	0	12.984375	0.015625	13
GodFatherPtOne	24029	16	0	5.3125	0	5.3125
GodFatherPtOne	24029	24	0	5.546875	0	5.546875
GodFatherPtOne	24029	32	0	9.0625	0	9.0625
GodFatherPt1andPt2	65,168	16	0	9.375	0	9.375
GodFatherPt1andPt2	65,168	24	0	5.84375	0.015625	5.859375
GodFatherPt1andPt2	65,168	32	0.03125	3.046875	0	3.078125

Figure 2 - File Encryption Time in Seconds



File Name	AES Decryption Time	RSA Decryption Time	SHA-512 Decryption Time	Total Decryption Time	Send Time			Recieve Time		
					File 1	File 2	File 3	File 1	File 2	File 3
TestText	0	0.25	0	0.25	0	0	0	0	0	0
TestText	0	0.21875	0	0.21875	0	0	0	0	0.015625	0
TestText	0	0.21875	0	0.21875	0	0	0	0.015625	0	0
100Words	0	0.234375	0	0.234375	0	0	0	0	0	0
100Words	0	0.234375	0	0.234375	0	0	0	0	0	0
100Words	0	0.21875	0	0.21875	0	0	0	0	0	0
1000Words	0	0.234375	0	0.234375	0	0	0	0	0.015625	0
1000Words	0	0.21875	0	0.21875	0	0	0	0	0	0
1000Words	0	0.21875	0.015625	0.234375	0.015625	0	0	0	0	0
5000Words	0	0.21875	0	0.21875	0	0	0	0	0	0
5000Words	0.015625	0.21875	0	0.234375	0	0	0.015625	0	0.015625	0
5000Words	0	0.21875	0	0.21875	0.015625	0.015625	0	0.015625	0.015625	0
10000Words	0	0.21875	0	0.21875	0	0	0	0	0	0
10000Words	0.015625	0.21875	0	0.234375	0	0	0	0	0	0
10000Words	0.015625	0.203125	0	0.21875	0	0	0	0	0	0
GodFatherPtOne	0	0.203125	0	0.203125	0	0	0	0	0	0
GodFatherPtOne	0.015625	0.203125	0	0.21875	0.015625	0	0	0	0	0
GodFatherPtOne	0.015625	0.21875	0	0.234375	0	0	0	0	0	0
GodFatherPt1andPt2	0	0.21875	0	0.21875	0.015625	0	0	0	0	0
GodFatherPt1andPt2	0	0.21875	0	0.21875	0	0	0.15625	0.015625	0	0
GodFatherPt1andPt2	0	0.234375	0.015625	0.25	0.015625	0	0	0.015625	0	0

Figure 3 - File Decryption and Transfer Time in Seconds

VI. Societal Impact

This project through a legal lens is subject to data protection laws. Depending on how the system is applied to an organization or the contents being encrypted, HIPPA and FERPA laws may be in effect. Through an ethical lens, the system only accesses text documents given, reads the documents, encrypts the documents and then deletes to documents to avoid abuse of access privileges and preserve information privacy. On the server side, when a client requests the return of their documents, all traces of the documents are deleted, and no records of client connection are kept.

VII. Contribution to Society

This project contributes a safe and secure remote file storage alternative. Due to the system being uniquely designed with the use hybrid encryption, the security of the files is a step above a system that uses a single encryption algorithm. Users may use the system to momentarily free up space on their computers or secretly store private files while knowing their information won't be accessed by a malicious third party. Drawbacks with the system in its current design reside in the limitation to only text documents and limited remote access.

VIII. Conclusion

Discovering a current and unique security issue was challenging and required creative thought and design. This project bridged the gap between practical computer engineering aspects presented in multiple undergrad courses and security principles taught in graduate courses. Aspects of the system, including application of encryption algorithms, TCP socket connection and wireless information transfer, required further research outside of the CSE curriculum but were nonetheless correctly implemented due to the foundation received through various courses. The system has a wide range of potential additions that may further challenge the foundation provided by the CSE curriculum such as secure database design and secure web application design.