Desgining a Remote File Storage System With the Help of Kerberos

Pete Alessandri Jr.
CSC 479
North Centeral College
Naperville, United States
palessandri@noctrl.edu

Luis Gonzalez
CSC 479
North Centeral College
Naperville, United States
lagonzalez@noctrl.edu

***Abstract*-The need for having a secure a remote file storage system is becoming more and more popular. Especially with people who do not want to store files on their local machine, so they look for some where remotely to store them securely. This paper will introduce our design of a remote file storage system that allows a user to store a file on a remote system while preserving the confidentiality and integrity of the file. Our design has three key parts: keeping the confidentiality of the file, maintain the integrity of the file, and finally making sure that only people with authorized access can access the remote file storage server.**

To keep the confidentiality of the file we used AES-256 encryption to encrypt the file before it is sent to the remote server. Before we encrypt the file, we ran it though a hash function to help preserve the integrity of the file. To do this we used SHA-3 512 with a counter. Now to transport the file we used a version of the Kerberos protocol.

I. INTRODUCTION

Technological advances in computing and the increase in hacking into devices and networks has led to more people storing their files on a secure remote file storing system. Over the past 5 years, cloud/remote storage systems have grown exponentially with more and more business moving away from traditional storage solutions with remote work becoming more popular. With such an increase in demand for remote storage systems is has become clear that we need to ensure that during the whole process the data is kept confidentiality, maintains integrity, and ability to safely transport the data to the remote storage.

The increase in popularity has made remote file storage systems a big target for hackers to obtain companies files and data. Thus, the need for strong user authentication, integrity checks, and maintaining confidentiality. While there are already many solutions, products/services already out there that provide companies and business a secure remote file storage system, we came up our own basic remote file storage system, using a hashing function, an encryption scheme, and a version of the Kerberos protocol.

II. SENDING A FILE FROM THE CLIENT TO THE REMOTE FILE STORAGE SERVER

A. Hashing the File

We used SHA-3 512 to hash the file that the client wants to send to the remote file storage server. So, the client first selects which file they wish to send to remote storage server and the first thing we do is run the file though a hash function. It looks something like this **H ({F | counter}) = h**. So, the F stands for the file that is being passed into the hash function which is H. We also pass a counter into the hash function as well to help against replay attacks, and with all of that we get the hashed file which represented by h. One of the reasons why we choose to hash the file with SHA-3 512 because it is still very secure and would require a lot of money and time to break it. Also, by hashing the file, it helps maintain the integrity of the file which is very important to us in our design.

Fig. 1. Client A selecting a file to be hashed.

B. Encrypting the Hashed File

After we hashed the file, we also must encrypt the file as well in order to keep the file confidential. We decide to encrypt the file with AES-256 due to the fact that it is one of the most secure encryption algorithms out there and as of right now it has not been cracked yet. Just like the hashing algorithm we used earlier it would require a significant amount of time and computing resource to break it thus making it an excellent choice to encrypt the client's file. So its going to look something like this: **E(F) = C**. With E being the AES-256 function in which we pass F (file) into it to get C (ciphertext). After encrypting the file, we now know that the file is confidential, and we do not have to worry about just anyone being able to open the file to see contents of it.

Fig. 2. Client A encrypting selected file.

C. *Client Accessing Remote File Storage Server*

After the client has hashed and encrypted the file that they want stored on the remote server, they must now request access to file storage server. To do this the client requests permission from the authentication server to have access to the storage server. We did this by incorporating a version of the Kerberos protocol with a ticket granting server. The first step is the client asking the authentication server to have access to the remote storage server. The authentication server replies with a session key and some session tickets as well. Which would look something like this: **{ $K_{A,TGS}$ }K_A || $T_{A,TGS}$ = A || Address || Δtime$_i$ || $K_{A,TGS}$ } K_{TGS}.** Once the client receives the session key and tickets, the client replies back to the ticket granting server with the authenticators and a ticket-granting ticket. Which looks like this: **File storage server || {A||time$_1$ } $K_{A,TGS}$ || $T_{A,TGS}$.** Once the ticket granting server receives the authenticators and a ticket-granting ticket, it decrypts the ticket-granting ticket with a shared key from the authentication server and. The ticket granting server then sends a Kerberos token back to client with another shared key with ticket-granting server and the file storage server. It would look like this: **A || { $K_{A,File\ Storage\ Server}$ } $K_{A,TGS}$ || $T_{A,File\ Storage\ Server}$ = { A || Address || time$_j$ || $K_{A,File\ Storage\ Server}$ } K_B.** Upon receiving the Kerberos token and another shared key, the client then sends a request to the remote file storage server with encrypted Kerberos token. Which looks like this: **{ A || time$_2$ } $K_{A,File\ Storage\ Server}$ || $T_{A,File\ Storage\ Server}$** Then the remote file storage server allows the client to have access for a certain amount of time that is specified in the token. Which would look like this: **{ time$_2$ } $K_{A,File\ Storage\ Server}$.** By using Kerberos, we are solving the issue of making sure only authorized users have access to the remote file storage server, which was one of our main goals in our design.

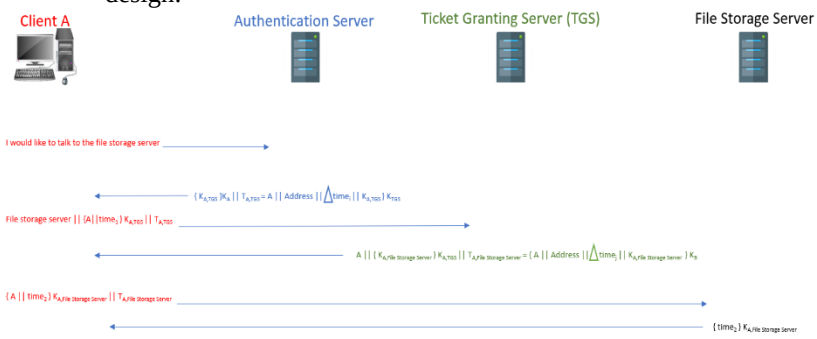
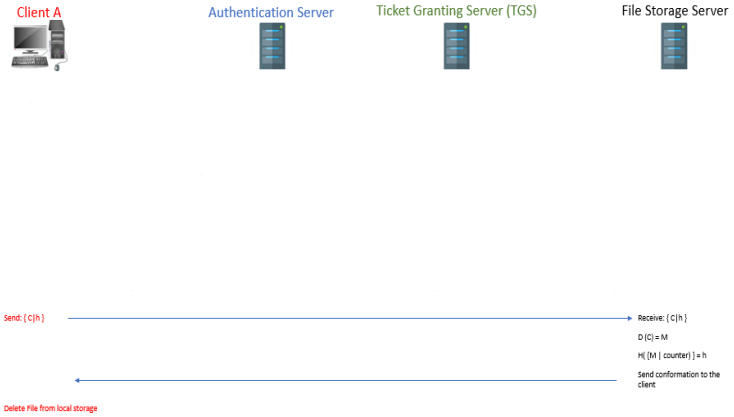


Fig. 3. Client gaining access to the remote file storage server.

D. *Sending the File to the Remote Storage Server*

Now that the client has been authorized to access to the remote file storage system, it will send the selected file that has been ran though the hash and has been already encrypted. Which looks something like this: **Send: { C|h }**. Once the remote file storage server receives the ciphertext and the hash it begins to decrypt the ciphertext with the shared key that was established out-of-band. Which it would look like this: **D (C) = M**. Next after the file storage server runs the file though the hash with a counter to if it matches what was sent from the client. Which would look like this: **H({M | counter) } = h**. After the storage server has confirmed the integrity of the file it will store the file on its local storage and call the file the date and time that the file was received. For example, if the file was received on March 15th 2021 at 6:33 pm then the file will be named 03/15/2021-6:33pm Finally, once the file storage server



confirms that the integrity of the file, it will send a conformation message back to client informing them that they received the file. Now once the client receives the conformation message the client will delete the file from local storage.

Fig. 4. Client sending encrypted file to the remote file storage server.

III. PUTTING EVERYTHING TOGETHER

Now that we explained how everything works now it time, we put all together and see what it looks like.

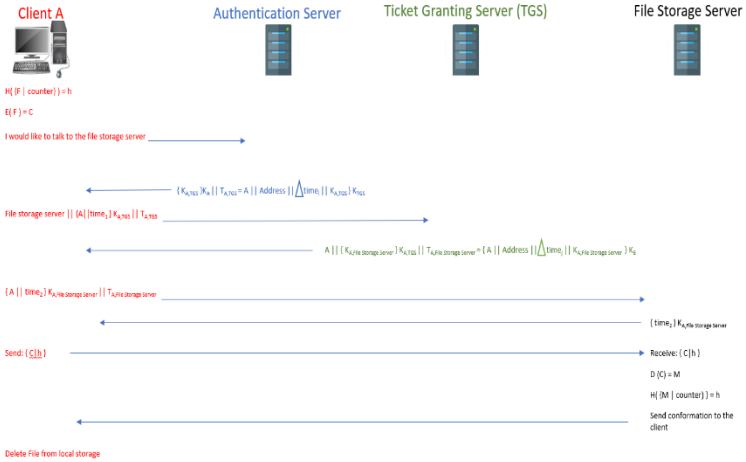
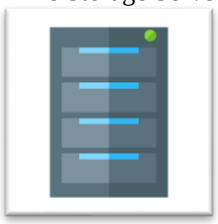


Fig. 5. Everything put together (The whole thing from start to end).

IV. CLIENT RETRIEVING FILE FROM THE REMOTE FILE STORAGE SERVER

File Storage Server



A. *Establish communication with the Server*

To retrieve file from the Server, the client must re-establish communication with the Server. The Client must confirm credential each time it needs to connect to the Storage Server with a new session key. This is needed to reduce the possibility of attacks. The Server must be able to confirm the credentials of the requester to send confidential information across the network. To achieve this, the Client has to go through the Kerberos protocol again as explained in *C* of Part II.

The first step is the client asking the authentication server to have access to the remote storage server. The authentication server replies with a session key and some session tickets as well. Which would look like this: $\{K_{A,TGS}\}K_A \| T_{A,TGS} = A \| Address \| \Delta time_i \| K_{A,TGS}\} K_{TGS}$. Once the Client receives the session key and ticket, the Client sends back to the ticket granting server with the authenticators and a ticket-granting ticket. Which looks like this: **File storage server** $\| \{A \| time_1\} K_{A,TGS} \| T_{A,TGS}$. Once the ticket granting server receives the authenticators and a ticket-granting ticket, it decrypts the ticket-granting ticket with a shared key from the authentication server and TGS. The ticket granting server then sends a Kerberos token back to client with another shared key with ticket-granting server and the file storage server. It would look like this: $A \| \{ K_{A,File\ Storage\ Server} \} K_{A,TGS} \| T_{A,File\ Storage\ Server} = \{ A \| Address \| \Delta time_j \| K_{A,File\ Storage\ Server} \} K_B$. Upon receiving the Kerberos token and another shared key, the client then sends a request to the remote file storage server with encrypted Kerberos token. Which looks like this: $\{ A \| time_2 \} K_{A,File\ Storage\ Server} \| T_{A,File\ Storage\ Server}$

Then the remote file storage server allows the client to have access for a certain amount of time that is specified in the token. Which would look like this: $\{ time_2 \} K_{A,File\ Storage\ Server}$. Once Client is authenticated through the Kerberos protocol, the Server will be allowed to continue the process of retrieving a file to the Client.

B. *Hash the file for integrity*

Now since the Storage Server has authenticated the requester, it is ready to send the file. However, before we send the file over, we need to hash the file with SHA-3 512. So, once the remote storage server knows what file to send back to the client, it is going to run the file through a hash function. Looks something like this: **H ({F | counter}) = h.** The F stands for stands for the file that is being passed into the hash function which is H. We also pass a counter into the hash function as well to help against replay attacks, and with all of that we get the hashed file which represented by h. Again, one of the reasons why we choose to hash the file with SHA-3 512 because it is still very secure and would require a lot of time and money to break it. Also, by hashing the file, it helps maintain the integrity of the file which again is a very important key to us in our design.

C. *Send file to Client.*

At this point the Client has been authenticated through Kerberos protocol and the integrity of the file has been confirm. It will encrypt the file using the AES-256 because it is one of the most secure encryption algorithms out there and as of right now it has yet to be cracked. Encrypting the file is going to look like this: **E(F) = C**. The E being the AES-256 function in which we pass F (file) into it to get C (ciphertext). Now that we have encrypted the file, we know that file is confidential, and we do not have to worry about just anyone being able to access the file. Now that the file has been hashed and encrypted it is ready to be sent to the client. It's going to look like this: **Send: { C|h }**. Now once the client receives the ciphertext it must decrypt it. Which looks like this: **D (C) = M**. Once the client has decrypted the ciphertext, it must next run the file though the hash in order to be able to read the contents of the file. Which will look like this: **H({M | counter) } = h**. Finally, once the client has confirmed the integrity of the file it will send a conformation receipt to remote file storage server.

D. *Confirm and Delete.*

Upon the receiving the receipt of delivery from the Client, the Storage Server will then confirm the time stamps of the receipt of delivery. If the time stamps match up, the Storage Server will be authorized to delete the file from local storage. However, if the time stamps do not match up, the Storage Server will not be authorized to delete the file in the case of a man in the middle attack.

V. PUTTING EVERYTHING TOGETHER

So, our design is very simple and basic one. Both process of sending and receiving the file are very similar to each other. The process of sending a file to the remote storage server starts with the client hashing and encrypting the selected file to be sent to the storage server. Then the client goes though the Kerberos protocol to gain access to the storage server. Once the client has be authorized access the client will send the ciphertext along with the hashed file to the storage server. Once the storage server receives the file it will begin to decrypt the file and hash it. Do this will help maintain the integrity and confidentiality of the file while in transit. Once that is done the storage server will send a conformation message back to the client, so that the client can delete the file from its local storage. Now its basically the same but in reverse if the client wants to retrieve a file from the storage server. This time around once

the client as be authorized by the authorization server, the storage server begins to hash and encrypt the file that is to be sent to the client. Once the file has been hashed and encrypted the storage server will send the file to client. Then the client will begin to decrypt and hash the file. Upon completion of that the client will send a conformation message back to the remote storage server so that it can delete the file from its local storage. This protocol describe above will fulfill the main requirements of a secure storage protocol over a P2P network. This protocol will achieve authenticity, integrity, authentication of file storing.

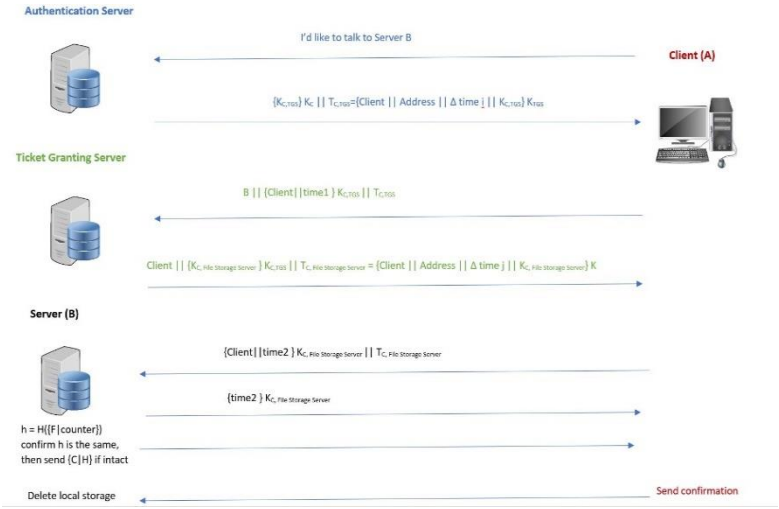


Fig. 6. Everything put together (The whole process of retrieving file from server).

VI. DESCRIBING OUR IMPLEMENTATION

A. Sending File to Server

So, for our implementation where the client is sending a file to the server, we start by giving the user three options to start with. The first option that the user has is to upload a file to server. The second option the user has is to download a file from the server. The third and final option is to exit the program. If the user decides to upload a file, the file is hashed using SHA 1 and encrypted using AES CTR mode to maintain security and integrity of the file. Once that is done the client will connect to the server over the network using socket programming. The client will display connected when it is connected to the server, then immediately send the encrypted file over to the server. On the server side once, it is connected to the client will display the IP address and port number. It will use the key that was established out of band to compare the two hashes to confirm that the file has not been tampered with. Once the server has confirmed it, it will display it and let the client know. Once the client receives conformation from the server it deletes the file from its storage and display file has been deleted. Now with everything being done the user can enter 3 to quit the program.

B. Retrieving File from Server

For our implementation of the server sending the file back to client begins with the user selecting what to do. The first thing would be to start to server to wait for the client to connect. Next the user would select to download a file from the server, which at that point the client would connect to the server over the network. The server will then hash the file and encrypt the file to keep the integrity of the file. Then compare the hash to the hash that was sent over with the file to confirm that the file is not corrupted. The server will then send the encrypted file over the network using socket programming to the client. Once the client receives the file it will hash it and decrypt it. The client will compare the two hash to confirm that the file has not been tampered with. Thus, sending a conformation message to the server to delete the file from its storage.

VII. SYSTEM TESTING

A. Testing Results

So, we ran two systems tests, the first one to test to see if the server could properly check to see if the file is intact before sending it back to client. So, we have the server compare hashes to confirm that the file is intact before sending to the client. If the hashes do not match it will not send the file to the client and will display file has been tampered with. When we ran this test, we purposely messed with the file to see if the server would catch that it was tampered with. Which it did, the server recognized that the file has been tampered with and displayed such and did not send the file to the client. The second test that we ran was to test to see if the client can check to see if the file has been tampered with. We solved this by having the client also compare hashes to make sure that the file was untampered with. When we ran this test, we purposely messed with the contents of the file to see if the client saw that it was messed with. When we ran test the client indeed caught that the file was tampered with and displayed so.