Project: Phase 4

Conor Lamb, Nick Mullen, Riley Marzka

Introduction

To protect against the threats detailed in this phase of the project, we chose to apply several ideas to securing our system. To satisfy T5, all of our encrypted messages across the system will contain a timestamp. The timestamp will allows us to detect message reorder and replay. We will store the timestamp of each message, and if any subsequent message contains a timestamp that is less than or equal to the last one, then the message is known to have been replayed.

To satisfy T6, all of the files within the system will be encrypted using 128-bit AES keys in GCM mode. 128-bit AES provides ample security, integrity, and fast, efficient operations.[1] Our AES cipher will be implemented in the GCM because it is an efficient block chaining mode and also utilizes GMAC for integrity. The keys used to encrypt the files will be generated by the GroupServer. One key will be generated for each group and will be used to encrypt the files viewable by that group. Upon a user authenticating into the GroupServer, he will be passed the keys for each of the groups he belongs to. The FileServer will not have access to the group keys and will only store files which have already been encrypted.The files will have been encrypted on the Client's side before transmission. The FileServer then will not have access to the contents of the files, nor will any unauthorized parties to which files are leaked by a malicious server. Each time a user is removed from a group, a new group key will be generated by the GroupServer. The administrator which removed the user will then use the new key to re-encrypt all of that group's files on the FileServer and delete all of the old files. The client will contact the GroupServer to update all of the group keys after every operation.

To satisfy T7, all tokens will be one-time use. To accomplish this, a random 1024-bit BigInteger will be added to each user's token and stored on the server, indexed by username. Using a 1024-bit BigInteger ensures that the numbers are sufficiently random and difficult to guess or to forge. Each time a user asks a FileServer to perform an operation, he will pass the FileServer his token. Upon initial connection, the FileServer will exchange a 128-bit AES session key in GCM mode with the GroupServer, via Diffie-Hellman. We chose Diffie-Hellman due to the fact that it involves 2 parties in the creation of a secret as well as provides confidentiality of said secret over a public channel. The Diffie-Hellman keys will be 1024-bit, which is sufficient to generate secure keys reasonably quickly.[2] The file server will then encrypt the token it received from the user and send it to the GroupServer. The GroupServer will then decrypt the token and compare the BigInteger value stored within the token to the one it has stored for the owner of the token. If they match, then the GroupServer will tell the FileServer that it is valid, otherwise it will say it is invalid. After this verification has occurred, the GroupServer will will generate a new random 1024-bit BigInteger and update the database and the client will need re-log in to the GroupServer before connecting to another FileServer. With this mechanism, we ensure that any unauthorized party who receives a stolen token from a malicious FileServer will not be able to use it.

Our system protects against the three threat models by adding timestamps to messages to prevent replay, encrypting all files using group keys, and by making tokens one-time use.

T5: Message Reorder, Replay or Modification

Threat Description

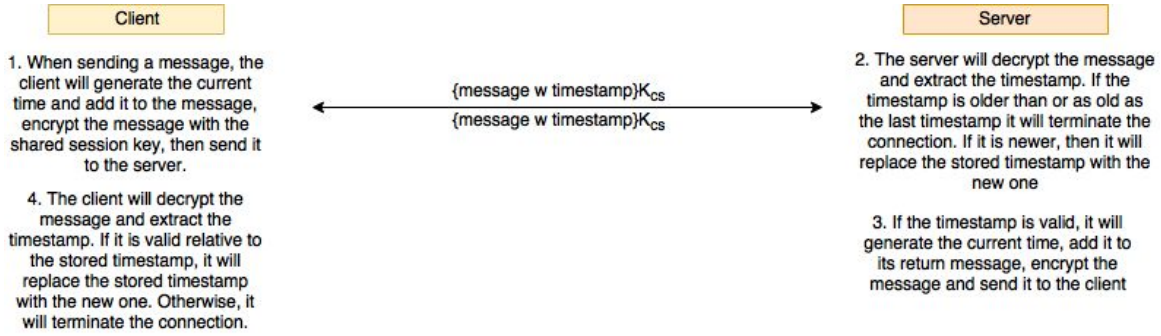
An active attacker could reorder, replay, or modify messages that they observe and/or intercept over the channel between the user and the server. This would allow the attacker to manipulate the server to carry out actions that have already been performed for an authenticated user but for a non-authenticated user by recycling the message, or trick the server into performing entirely new operations without the appropriate permissions by modifying messages.

Solution

To ensure that the server and the client are both receiving unmodified messages in the intended order, we will make several modifications. To prevent message reorder or replay, we will add a timestamp to each message that is sent between the client and the server. The timestamp will be encrypted using the shared session key so that it will not be visible to an outside attacker. After the key exchange is complete, the client and server will both record the time of the first message received. Then, if any subsequent message carries a timestamp that is older than or as old as the last message received, it will deny the message as it is obviously not a fresh message if a new message is timestamped as being older than or generated at the same time as the previous one. Upon message denial, the connection between the server and the client shall be terminated.

To prevent message modification, we will use the same method from our previous phase in T4. Every message to and from the server will be encrypted with a shared 128-bit AES-GCM session key. Using GCM mode ensures that the message will not be modified, as it not only encrypts the message but includes an authentication tag to ensure the message's correctness. If a message has been modified, then the shared key will not be able to decrypt the message, which would alert the server or client that the message has possibly been modified and it should terminate the connection.

Diagram



Correctness

Our solution relies on several assumptions. First, that the shared key between the server and client has not been compromised. Second, that the server and client both have synchronized clocks. If the server and client are not synchronized, then a message can appear to have been replayed, when it is actually fresh, or vica versa.

T6: File Leakage

Threat Description

Since file servers are untrusted, there exists a possibility that file servers may leak files from the server, allowing anyone to obtain them. In our current implementation, if files were to be leaked from the server, anyone would be able to read them as the files are unencrypted and are only secured under the assumption that the file servers will only distribute files to properly authenticated users via tokens with the appropriate permissions. If sensitive data is stored in these files, it is important that only authorized users are able to gain access to the information contained in the files, and any unauthorized party who obtains the file should not be able to gain access to the file's contents.

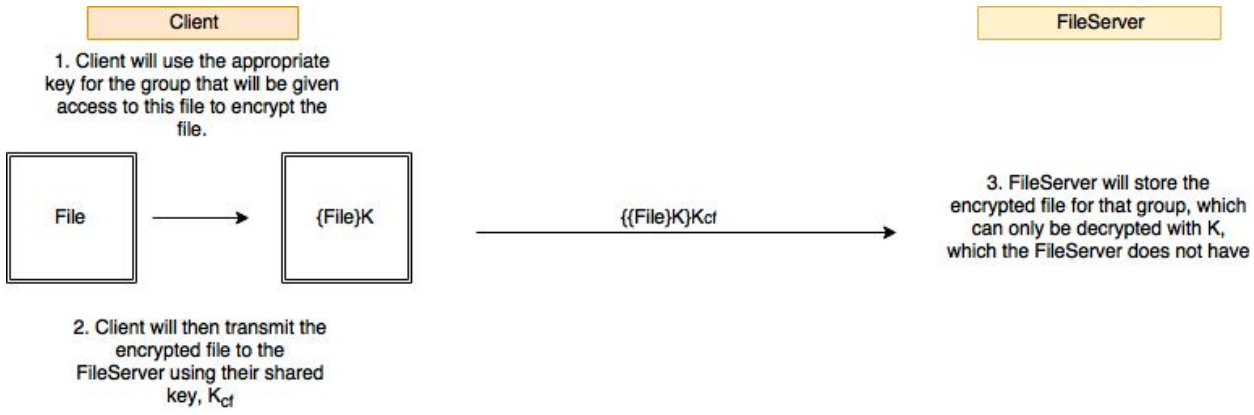
Solution

To ensure that information contained in files is viewable only by the people who are authorized to view them, we will implement encryption of individual files using 128-bit AES keys. There will be an encryption key that is stored on the GroupServer for each group that is registered within the system. Whenever the user logs into and properly authenticates itself with the GroupServer, the GroupServer will pass the keys to the user for each group that they are a member of. Before uploading any files to a FileServer, the client will encrypt the files using their group's shared encryption key. Then, whenever they've download files from a FileServer, the user will use their group keys to unlock those files. The keys will remain on the client and be passed to the client from the GroupServer, ensuring that the FileServer will never have direct access to a key, which ensures that the FileServer cannot decrypt files before leaking them or distributing the keys along with the files.

Having separate keys for each group ensures that only the members of a group can read the files contained for that group. Whenever a user is removed from a group, which can only be done by the creator of that group, the client will automatically download every file from that group from a FileServer, decrypt the files with the old encryption key, contact the GroupServer for a new encryption key, re-encrypt and then reupload the files to the FileServer. This ensures that whenever a user is removed from a group, they will no longer be able to access files leaked from a FileServer that belong to their former group. All clients will periodically (after each operation) check with the GroupServer to ensure their keys are up to date.

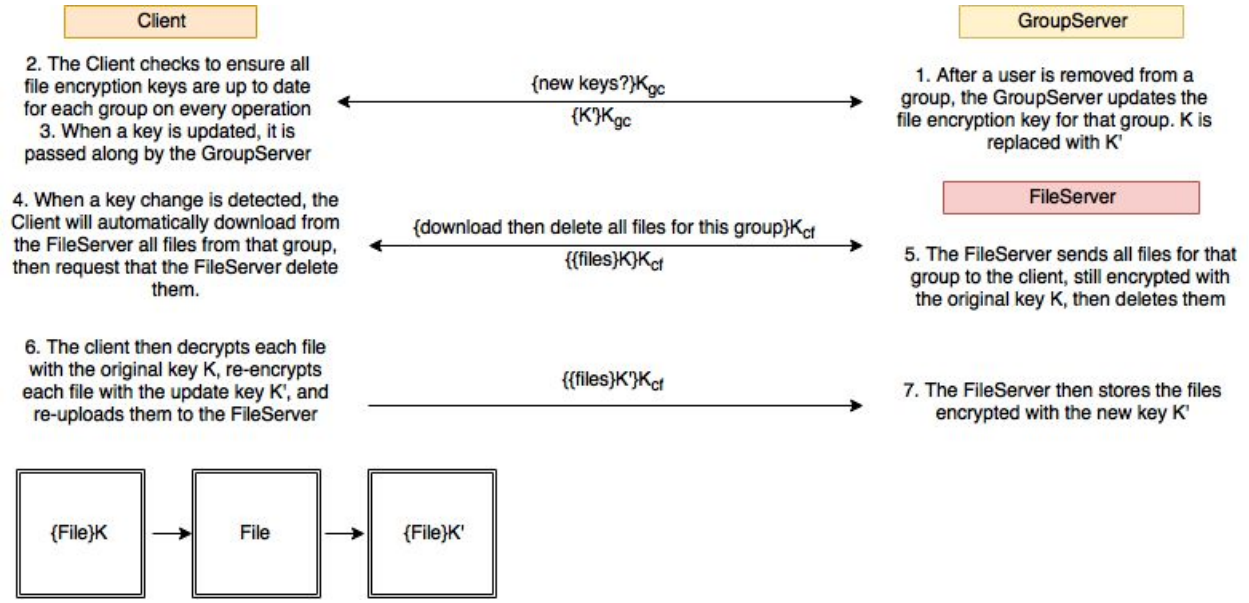
Diagrams

Encrypting and Uploading a File



Downloading and Decrypting a File

Removing a User From a Group



Correctness

Our model does several things to protect against file leakage including storing encryption keys only on the client side and the GroupServer, thus the FileServer does not have access to the encryption keys to decrypt files before they are leaked. Thus, leaked files are guaranteed to be decrypted only by keyholders. By changing the keys whenever a member is removed from a group, it is ensured that only current group members can decrypt files belonging to that group.

Our model assumes that the GroupServer and client have been properly authenticated and their communication is encrypted using a shared single session key. It also assumes that users will not leak their file encryption keys so that only trusted group members can decrypt files.

T7: Token Theft

Threat Description
Since FileServers are untrusted, it must be expected that a malicious FileServer may substitute tokens for various users; for example, a FileServer may give an administrator token to every user, regardless of the user's actual permissions. If a user who is not an administrator is given administrator privileges, then that user may download files that they should not have access to, or delete files that they should not be authorized to delete. It must be expected that token theft may occur, and we should mitigate this throughout the system.

Solution
In our implementation, the Client requests a new Token from the GroupServer after every operation is performed as the Client wants to have the most up-to-date Token for its user at all times. Thus, since the Client is constantly requesting new Tokens from the GroupServer, we will make the Tokens one-time use. Anytime a FileServer is passed a Token, it must validate that Token with the GroupServer, which will check to ensure that Token has not been used before. If it has not been used before, it will return to the FileServer that it is a validated Token.

The FileServer and the GroupServer will perform a Diffie-Hellman exchange to exchange symmetric 128-bit AES-GCM keys using the same procedure described in T1 of this project. This shared key will encrypt all communication between the GroupServer and FileServer to ensure that their interaction is confidential.

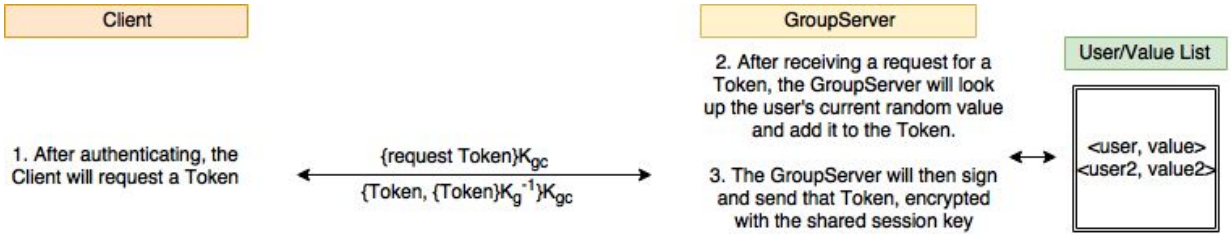
The GroupServer will ensure that Tokens are only used one time by including a field in the Token which is a random 1024-bit BigInteger; the GroupServer will keep a record of each username in the system and that user's current random value. Whenever a Token is requested, the GroupServer will issue the user a Token, with that BigInteger value. Whenever a Token is used, the GroupServer will check that the value equals the current value for that user. If it does, it will permit the operation, and then generate a new random value for that user for the next time that a Token is requested by that user.

This mechanism ensures that if a Token is stolen or leaked, it cannot be used as it can only be used one time.

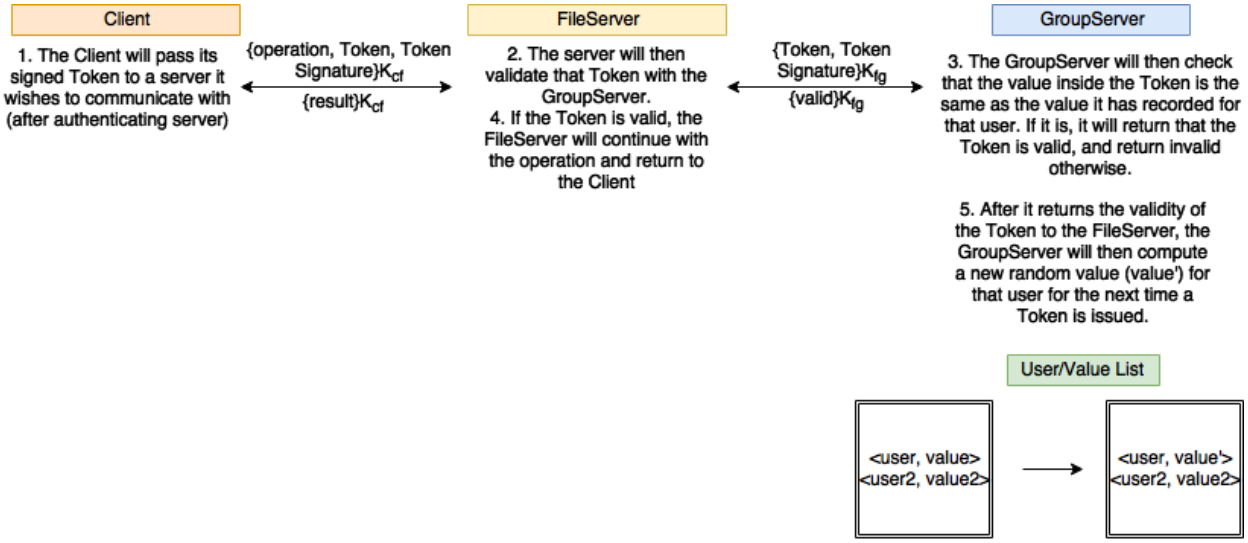
Diagrams

Requesting a Token

After a user has already been authenticated by the GroupServer, the user may request a Token from the GroupServer.



Using a Token



Correctness

Our model operates under the assumption that a FileServer will always verify a Token with a GroupServer before allowing an operation to proceed. It also assumes that the current BigInteger values stored on the GroupServer are not compromised. Also, the model assumes that all users have been properly authenticated by the GroupServer, and thus a user cannot impersonate another to be issued one or multiple Tokens by the GroupServer despite forging the identity of the proper Token holder.

Conclusion

The design of our system successfully protects against the threats outlined for this phase of the project, while maintaining simplicity as well as efficiency. Adding a timestamp to a message is trivial to both calculate as well as to encrypt. Nevertheless it allows us to detect and prevent replay attacks. By baking a GMAC into the AES encryption of all of our messages, we are able to detect message modification without any extra operations. By using AES we are able to efficiently encrypt all the files on the system. While it is a relatively expensive operation to re-encrypt all of the files when a user is removed from a group, it is a necessary measure. Also, it can be reasonably assumed that this operation will be infrequent. Finally, calculating and comparing random 1024-bit BigIntegers is inexpensive and the database used to store them will be relatively small, allowing us to easily make tokens one-time use. Simplicity, efficiency, and thoroughness guided our design process. Our system thoroughly protects

against all of the threats outlined in this phase of the project, while maintaining a high degree of efficiency. We tried to keep the system as simple as possible so as to minimize possible points for failure.

Sources

1. "Announcing the ADVANCED ENCRYPTION STANDARD (AES)" (PDF). Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001. Retrieved October 2, 2012.
2. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice" (PDF). Retrieved 30 October 2015.