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 an increment value, which will be generated by the server as a small, random integer value, that will be incremented each time a message is sent by either the Client or the Server. This prevents replay and reorder of individual messages because both parties will only expect the next message that comes in logical order from the increment value. Our use of a unique session key will prevent entire conversations from being replayed, and our use of 128-bit AES-GCM will ensure message integrity as GCM mode guarantees the contents of the message have not been modified. 128-bit is a sufficient key length for both efficiency and security, and GCM mode ensures authenticity.

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 and all new or modified files going forward will be encrypted with the new key, which ensures that only current group members have access to current files.

To satisfy T7, users will pass to the GroupServer the server name and port number of the FileServer they wish to connect to. The GroupServer will then incorporate that information into the Token and pass back the Token and its signature. When performing an operation with a FileServer, the Client must pass both the Token and its signature. The FileServer will verify both that the signature is valid, ensuring that the Token has not been modified, and verify that the provided server name and port match its own, ensuring that the Token is intended for use on this server.

Our system protects against the three threat models by adding increment values to messages to prevent replay and reorder, encrypting all files using group keys, and by signing Tokens for use on one FileServer.

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

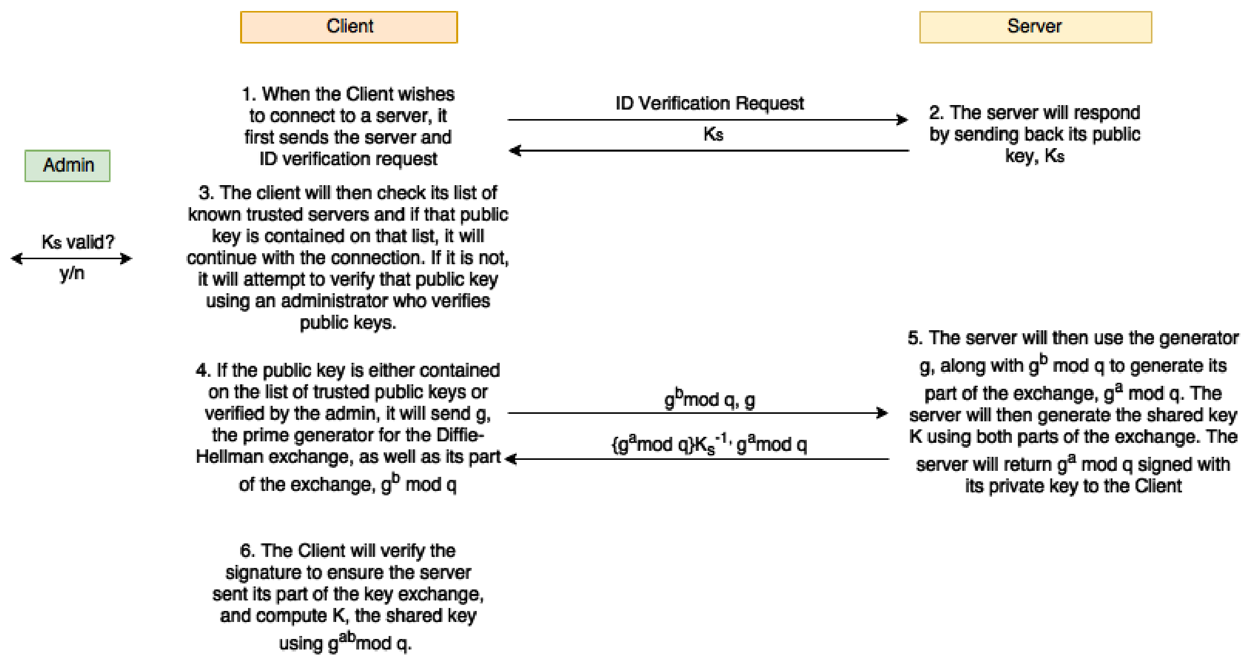
To ensure that the server and the client are both receiving unmodified messages in the intended order, we will make several modifications. First, we will modify our key exchange protocol from T1 such that the server will sign its part of the Diffie-Hellman exchange. Signing part of the Diffie-Hellman exchange will prevent a man-in-the-middle attack, as the attacker who intercepts the message would be unable to modify the message without invalidating the signature, thus the attacker would have to forward the unmodified message to the user. With this mechanism, an attacker would be able to hijack the connection to the server from the client. He can intercept the user's part of the exchange and substitute his own Diffie-Hellman key part. The attacker would then share a private key with the GroupServer. This threat is immediately mitigated, however, since authentication into the GroupServer requires knowledge of the user's password, which the attacker does not have. Optimizing this protocol also allows us to eliminate the random challenge encryption, because by signing the Diffie-Hellman exchange, the Client knows that the server is in possession of the private key.

To mitigate message replay and reorder, after the key exchange the server will send to the user a random integer, which will be used as an increment value. After receiving the increment value, the Client will then increment the value by one for each message it sends to the Server. The Server will check that it received messages in order from the Client and subsequently increment the value when it sends messages to the Client. This protocol ensures that the Client and the Server both receive messages in their intended order, which prevents message reorder and replay. Since the Diffie-Hellman created shared key is uniquely created with each session, this prevents an entire session being replayed as well.

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.

Diagrams

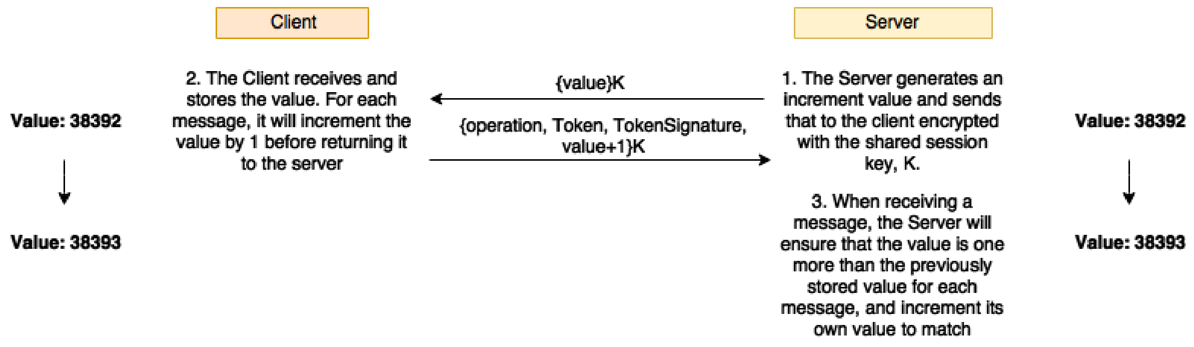
Key Exchange



Client
Server
Admin
1. When the Client wishes to connect to a server, it first sends the server and ID verification request
ID Verification Request
Ks
2. The server will respond by sending back its public key, Ks
Ks valid?
y/n
3. The client will then check its list of known trusted servers and if that public key is contained on that list, it will continue with the connection. If it is not, it will attempt to verify that public key using an administrator who verifies public keys.
4. If the public key is either contained on the list of trusted public keys or verified by the admin, it will send g, the prime generator for the Diffie-Hellman exchange, as well as its part of the exchange, g^b mod q
g^bmod q, g
{g^amod q}K_S^{-1}, g^amod q
5. The server will then use the generator g, along with g^b mod q to generate its part of the exchange, g^a mod q. The server will then generate the shared key K using both parts of the exchange. The server will return g^a mod q signed with its private key to the Client
6. The Client will verify the signature to ensure the server sent its part of the key exchange, and compute K, the shared key using g^{ab}mod q.

Increment Value

Directly following the key exchange, the server will send back to the client a random integer value to be incremented to determine the order of messages.



Client
Server
Value: 38392
Value: 38393
2. The Client receives and stores the value. For each message, it will increment the value by 1 before returning it to the server
{value}K
{operation, Token, TokenSignature, value+1}K
1. The Server generates an increment value and sends that to the client encrypted with the shared session key, K.
3. When receiving a message, the Server will ensure that the value is one more than the previously stored value for each message, and increment its own value to match
Value: 38392
Value: 38393

Correctness

Our solution relies on several assumptions. First, that the shared key between the server and client has not been compromised. Also, it assumes that the Client and the Server will not leak the increment value that is being stored, however even if it was, the messages are encrypted which would make knowing just the increment value relatively useless.

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.

Our policy for this threat states that if a user is a member of a group, then that user should be able to view, modify, and download all files belonging to that group. If a user is removed from a group, then that user should no longer have access on a FileServer to access any files from that group, but specifically, the user should not be able to view any file that has been modified or uploaded since they were removed from the group.

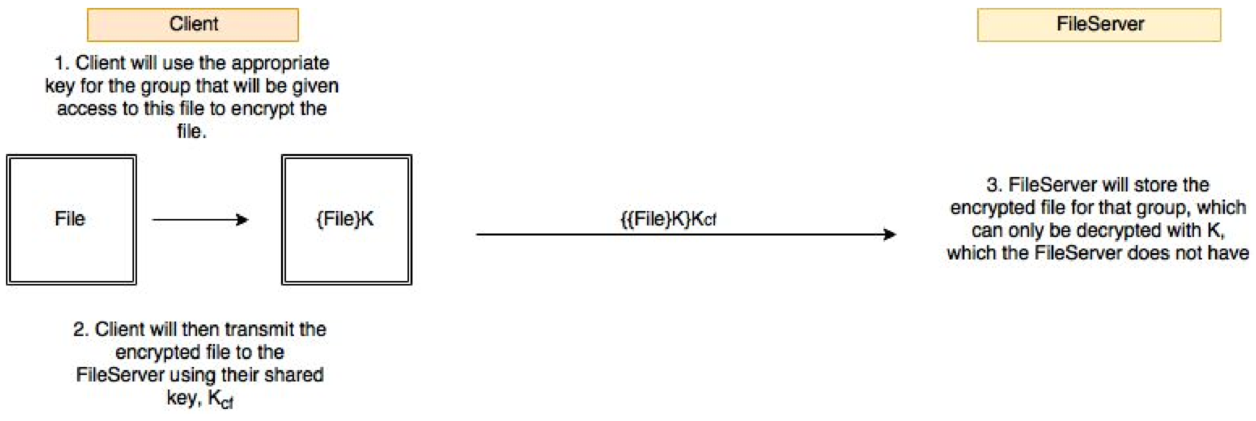
Solution

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 list 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.

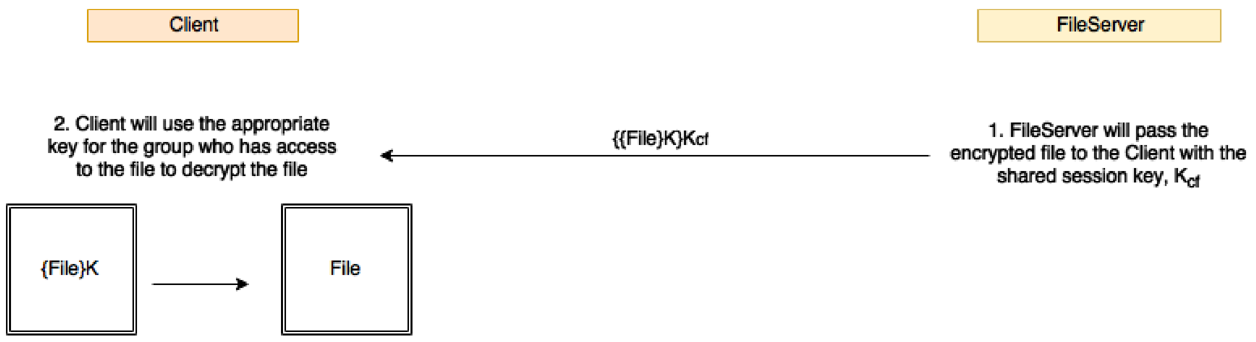
Having separate keys for each group ensures that only the members of a group can read the files contained for that group. Whenever a user is removed from a group, the key for that group will change, and that new key will be passed to every Client that is still a member of that group. Then, all files that are uploaded to a FileServer after the permissions have changed will be encrypted with the new key, K' which ensures that the user that has been removed will not be able to see the content of any file uploaded since they were removed, even if he were to gain access to the encrypted file through leakage. This strategy allows for a user to still be able to decrypt and read any file that was leaked to them that had not been modified since they left the group. This is still a secure strategy because that user would have been able to make a local copy of that file when they had access to the group, which means that the security of the group's files has not been compromised.

Diagrams

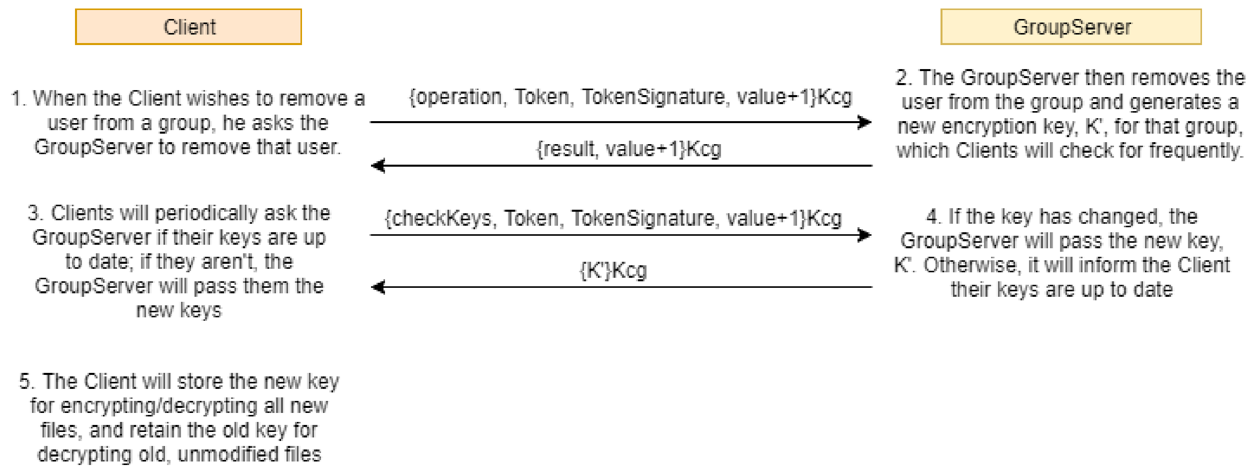
Encrypting and Uploading a File


Client
1. Client will use the appropriate key for the group that will be given access to this file to encrypt the file.
File
{File}K
{{File}K}Kcf
FileServer
3. FileServer will store the encrypted file for that group, which can only be decrypted with K, which the FileServer does not have
2. Client will then transmit the encrypted file to the FileServer using their shared key, K_{cf}

Downloading and Decrypting a File


Client
FileServer
2. Client will use the appropriate key for the group who has access to the file to decrypt the file
{{File}K}Kcf
1. FileServer will pass the encrypted file to the Client with the shared session key, K_{cf}
{File}K
File

Removing a User From a Group


Client
GroupServer
1. When the Client wishes to remove a user from a group, he asks the GroupServer to remove that user.
{operation, Token, TokenSignature, value+1}Kcg
{result, value+1}Kcg
2. The GroupServer then removes the user from the group and generates a new encryption key, K', for that group, which Clients will check for frequently.
3. Clients will periodically ask the GroupServer if their keys are up to date; if they aren't, the GroupServer will pass them the new keys
{checkKeys, Token, TokenSignature, value+1}Kcg
{K'}Kcg
4. If the key has changed, the GroupServer will pass the new key, K'. Otherwise, it will inform the Client their keys are up to date
5. The Client will store the new key for encrypting/decrypting all new files, and retain the old key for decrypting old, unmodified files

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.

By encrypting all new and modified files with a new encryption key after a user has been removed from a group, this ensures that even if the removed member gained access to a leaked file, they would not be able to decrypt it. Even though non-modified files are still encrypted with an old key, this presents no security risk as that user may have a local copy of that file anyway.

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.

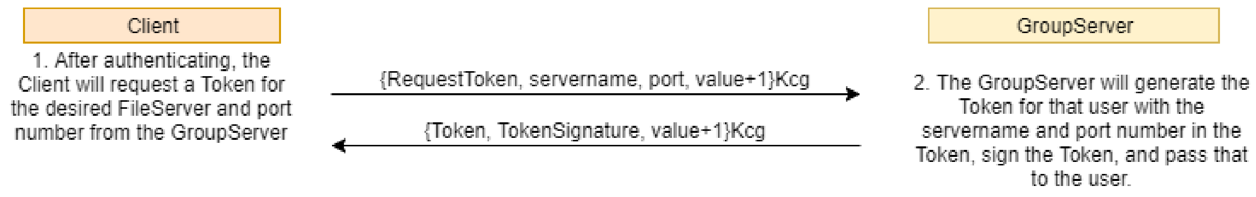
Our policy for this threat is that Tokens should be authorized to work with only one FileServer, and that each FileServer will only accept Tokens that are signed by the GroupServer and intended for its use.

Solution
Each time the user wishes to connect to a FileServer, the Client will pass to the GroupServer the server name and the port number of the FileServer they wish to connect to. The GroupServer will then issue the Client a Token with the desired port and server name of the FileServer as fields of the Token, and then issue the Client a signature of that Token. Whenever the user connects to a FileServer, the FileServer will verify the signature of the Token, then check if the Token is intended for use on this server by checking if the desired server and port number match. If they do not, the server will deny the operation.

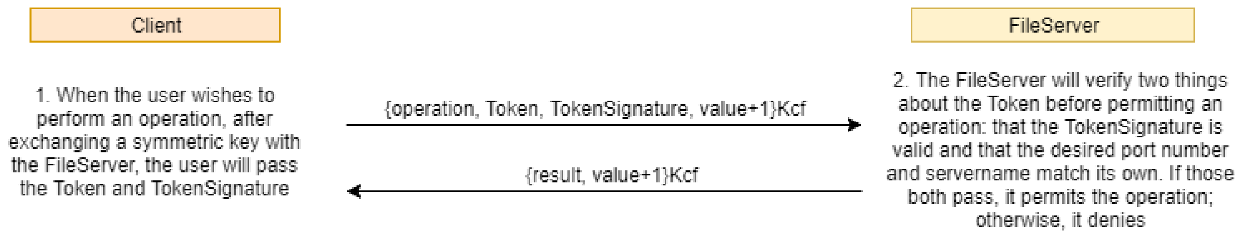
This ensures that Tokens leaked by one FileServer cannot be used by another, as the FileServer must check that the desired port number and server match its own. If the Token is modified to change the desired port number and server manually, then the signatures will not match and the operation will be denied. This ensures that leaked or stolen Tokens can only be used on the FileServer for which the server name and port number match.

Diagrams

Requesting a Token


Client
1. After authenticating, the Client will request a Token for the desired FileServer and port number from the GroupServer
{RequestToken, servername, port, value+1}Kcg
{Token, TokenSignature, value+1}Kcg
GroupServer
2. The GroupServer will generate the Token for that user with the servername and port number in the Token, sign the Token, and pass that to the user.

Using a Token


Client
1. When the user wishes to perform an operation, after exchanging a symmetric key with the FileServer, the user will pass the Token and TokenSignature
{operation, Token, TokenSignature, value+1}Kcf
{result, value+1}Kcf
FileServer
2. The FileServer will verify two things about the Token before permitting an operation: that the TokenSignature is valid and that the desired port number and servername match its own. If those both pass, it permits the operation; otherwise, it denies

Correctness

Our model operates under the assumption that a FileServer will always verify the desired port number and server name as well as the Token signature before permitting an operation. Our mechanism ensures that properly behaving FileServers will only allow Tokens intended for use on itself to be used, otherwise it will deny the operation.

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 an increment value 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.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.