# Project: Phase 3

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. First, all communication in the system will be encrypted with symmetric shared keys, exchanged using a Diffie-Hellman protocol. 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.[1] The keys will be generated randomly using a p and g that are randomly generated each time. The symmetric keys will be 128-bit AES keys in GCM mode. 128-bit AES provides ample security, integrity, and fast, efficient operations.[2] Our AES cipher will be implemented in the GCM because it is an efficient block chaining mode and also utilizes GMAC for integrity. These symmetric keys (session keys) will be exchanged in two places: between the client and the GroupServer (Kcg) and between the client and the FileServer (Kcf). Exchanging session keys on each connection guarantees that the session keys are not reused, which maintains forward secrecy because if a session key is compromised, only information from one session is insecure.

Each connection to a server is authenticated to the client using the public key cryptography. The GroupServer (GS) has a public key (Kg) and each file server has a public key (Kf). These public keys will be encoded using 2048-bit RSA which contemporarily provides a high level of security without compromising performance.[3] Clients will authenticate servers using their public keys, which will be displayed to the client on first connection and can be verified by the user by cross referencing the displayed server key (subsequent connections will be checked against a list of known trusted servers on a per client basis). The GS will authenticate the client using a username and password, which the GS will check against a database of usernames and password hashes to check if they match. If they do match, the GroupServer will issue the client a session encrypted Token and encrypted signed Token which the client can use to authenticate themselves to the GroupServer for further operations, as well as to any file server.

Every access to every operation is checked for permissions via Tokens. Tokens will be signed using the GS''s private key (Kg-1). Both GS and any FileServer (FS) will require both the Token signature and the Token from the client in order to verify the signature and confirm that the Tokens have not been modified since being originally issued by the GroupServer.

Our system protects against the four threat models by establishing connections in secure key exchanges, passing Tokens only to authenticated users, allowing users to authenticate servers using their public keys, and by encrypting all network traffic with per-session keys.

# T1: Unauthorized Token Issuance

**Threat Description**

To carry out any actions on our file sharing system, a user needs to have a token which contains his or her own group permissions for file sharing actions. With each action both, the group and individual file servers, check the user's supplied token to authenticate that a user is allowed to take said action. Our file sharing system relies on these token checks to grant permissions for each connected user, thus it is of utmost importance for the entity (our group server) that distributes tokens to users, always allocates the correct token (permissions) to the specific user who owns that token. A token mistakenly or insecurely obtained by a different user than the proper owner will grant improper user privileges to see, steal, and manipulate another user's confidential files as well as the possibility of modifying or deleting their group's status. In our former build, we implement a very weak form of authentication as well as transmit tokens without integrity or confidentiality. Our user authentication only asks for the name of a user before logging them in (authenticating them). This means that solely knowing a username (visible publicly to all peers in a group), grants anyone who knows the username the ability to log in as that user. Upon receiving a correct username, our build will then insecurely (in plaintext on a public channel without an integrity check) transmit the usertoken to the user. This means that an adversary can see the token and 1. Know all about a user's permissions. 2. Take a user's user token and impersonate them in our system 3. Manipulate the token that was sent to the user.

## Solutions

The group server, which allocates the tokens to users initially logging in, must authenticate that a user is the proper owner of that token and that the token is securely transmitted to them to avoid an adversary having access to a token he or she does not own.

### A. Correct Token Issuance (Authentication)

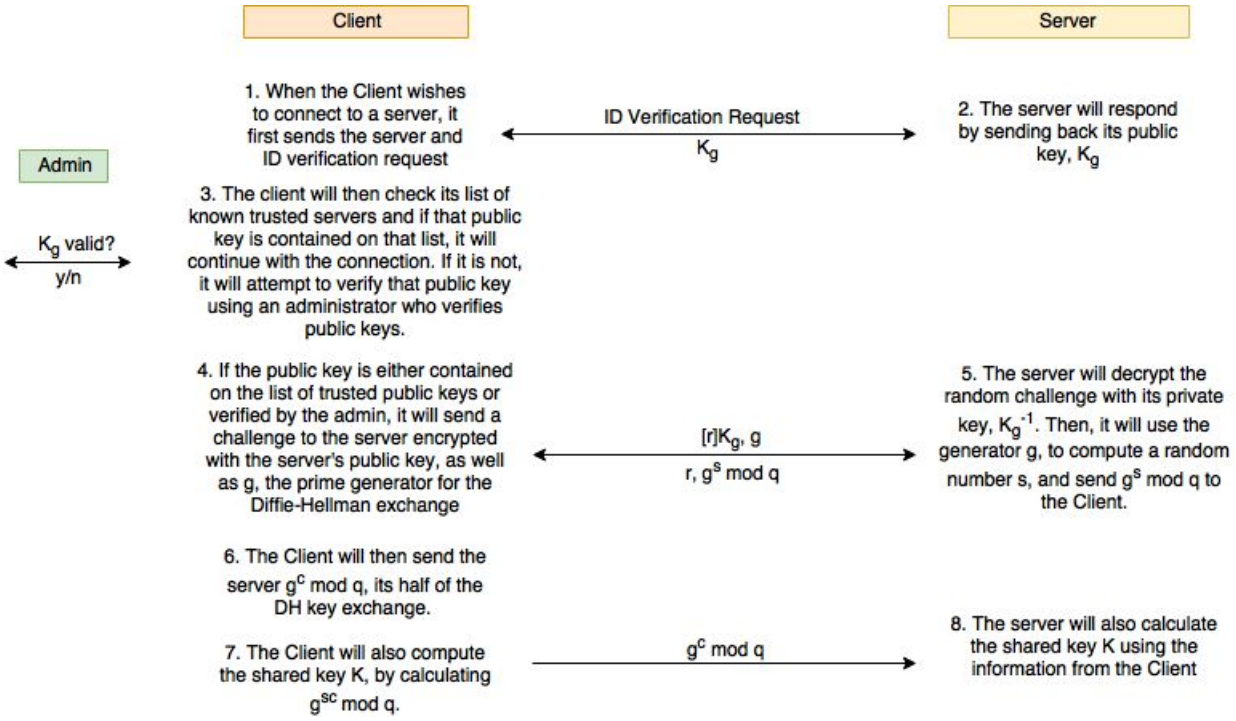
Issuing the correct token involves authenticating that we are communicating with the correct user. The authentication process will be based on a user submitting a username and correct password to the group server over an encrypted channel (*solution (b)*. The group server will then take the received password, hash it using the SHA-3 hash function, and check it against the hashed password associated with that user stored in our private database. The hashed stored password is a private attribute of the User account, stored in the UserList. If the hash matches, the user will be transmitted the corresponding token and thus his/her corresponding privileges, if the user provides an incorrect password or username, he/she will be disconnected

from the system after 5 attempts. The token will be signed with the server's public key to avoid forgability.

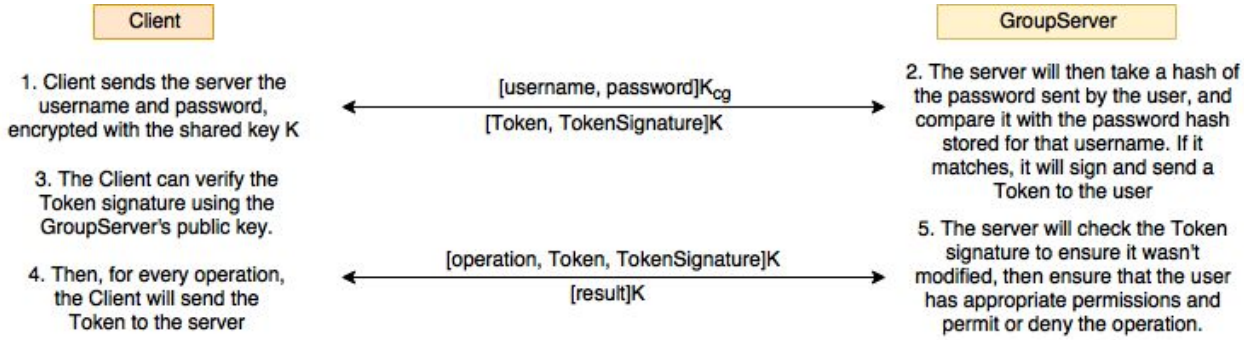
### B. Secure Token Transmission (Confidentiality and Integrity)

We need a secure channel with which to communicate and authenticate a user. This is because our authentication process (*solution (a)*) will communicate sensitive information over a public channel that must not be visible nor augmentable by outside parties. To accomplish this we first will implement the Diffie-Hellman (D-H) Secret Key Exchange Protocol to exchange a secret 128 bit *session key* with the client with which to encrypt our communication over a public channel. The session key is a 128-bit AES key operating in GCM mode which will, by the definition of GCM, provide strong contemporary security and immutable integrity.

First, the Client will authenticated the server by sending an ID verification request. The server will return its public key and the Client will compare that against a list of trusted servers it has connected to in the past. If this is the first connection to a server, it will verify the public key of the GS with the displayed public key via an admin. If the key is verified, the Client will encrypt a challenge with the server's public key and send that to the server. The challenge is a 1024-bit randomly generated BigInteger, which is a large enough random integer to guard against a possibly replay attack. The client will also send g, a D-H generator, and q, the modulus for the D-H exchange. The DH generator is defined by our Cryptographic Provider BouncyCastle which is assumed will be highly secure. The server will return the random challenge, verifying its identity as it has decrypted the challenge with its private key, with its part of the D-H exchange. The client will use that to compute the shared key, K, and send its part of the D-H exchange back to the server. Now, both parties are in possession of the shared key K.



After the key exchange has taken place, the server will then authenticate the client using a username and password, then issue the Client a Token. The Token signing procedure is explained in further detail in T2.




**Solution Reasoning**

We believe that our system aptly mitigates the risks associated with Unauthorized Token Issuance because at each step, confidentiality, integrity, and availability are aptly covered in accordance with contemporary information security standards.

Authenticating users (clients) relies on users having unique user names and secret passwords. If a user has knowledge of both of those pieces, we assume that we have the correct identity and allocate the token. The weakest point in our system would come from users with bad passwords. This can be mitigated by requiring the users to enter a strong password according to our own specifications. Passwords are also unable to be recovered from the client side if our system were to be compromised as we have a private database with only the hashes of passwords and not plaintext, thus preserving the user's passwords and foiling any adversary attempt of impersonating a user. We are also operating under the assumption that the group server is trusted and thus the user has the correct address of the server to communicate with.

The authentication process is also protected from passive and active attacks based on the encryption protocols in our communication. The initial interaction is a Diffie-Hellman Key Exchange which allows the server and the client to share a secret key without outsiders ascertaining the key. With that secret key, the client now sends the server his authentication credentials encrypted with AES-GCM. In other words, communication between the server and client will appear completely random to adversaries and if they try to augment it, either party will know.

# T2: Token Modification/Forgery

**Threat Description**

If a token is modified by a user, they can increase their access rights and violate established permissions. A modified token could allow an unauthorized user to add themselves to groups they aren't members of and upload and download files they otherwise wouldn't have access to. Modified tokens could also allow a user to make themselves the owner of a group, giving them the ability to add or delete users from groups, or entire groups themselves.

**Mechanism**

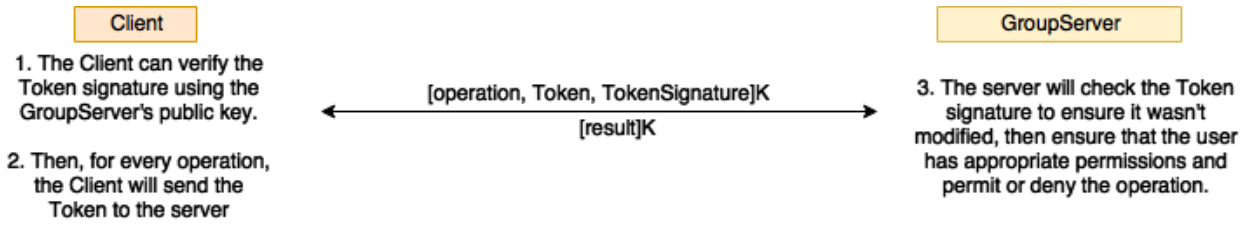
To implement protection against token forgery, we can modify our GroupServer such that after it has exchanged and agreed on a symmetric key with the client, it will use that shared symmetric key between the GroupServer and the client to pass the user's token, as well as a Token signature, which has been generated using the GroupServer's private key. The public/private keypair for the GroupServer is 2048-bit RSA which is sufficient for security against brute-force attacks, and efficient enough to complete operations quickly. The key exchange for the symmetric key and the symmetric key itself is described in further detail in T1.

The Token signature is computed by serializing the Token into a string that can then be converted into a byte[] array to be signed with the public key. The client can then verify the authenticity of the token by verifying the signature over the Token string with the GroupServer's public key, which is a publicly known key. Whenever the client wishes to pass that token to a FileServer, the FileServer can also verify the token's authenticity using the GroupServer's public key and the Token string.

Both the FileServer and GroupServer will require the signed Token for every access of an operation. This complete mediation protocol ensures that every access to the system is legitimate and will deny any illegitimate operation.

When the user has been authenticated, the GroupServer will generate a Token, then pass the Token and signature to the user as in the following diagram.


After the Token has been issued to the user, the user must pass the Token and the Token signature to either the GroupServer and the FileServer for each operation the user wishes to complete. The server will check the Token signature using the GroupServer's public key, then permit or deny the operation based on the validity of the signature, and the permissions contained in the Token.




**Correctness**

This method of ensuring against token forgery is correct with a few assumptions:

1. The GroupServer's private key has not been compromised
2. The verification of signatures is correct
3. The original key exchange was not compromised
4. The original key was not distributed

Requiring the Token signature to be verified on each access ensures that every access is valid and legitimate. If, upon decryption using the GroupServer's public key, the signed token does not produce a provided Token object, the server will reject the operation, ensuring that forgery of a Token will not allow privileges to be modified.

# T3: Unauthorized File Servers

**Threat Description**

Our trust model assumes that properly authenticated file servers are guaranteed to behave as expected. This trust model does not guarantee anything in regards to unauthenticated file servers. An unauthenticated file server may behave unexpectedly or maliciously when handling files, user tokens, etc. For example, an unauthenticated file server may corrupt files within the system. Beyond benign confidentiality and integrity concerns regarding file contents, an adversary could exploit this threat by corrupting files in such a way as to corrupt a user's machine or even corrupt the entire system. An unauthenticated file server could also be used to leak files to unauthorized users or other entities outside the scope of the system. Finally, an adversary could use an unauthenticated server to steal, view, and modify users tokens. The threats posed by a breach in token integrity has been discussed in previous sections.

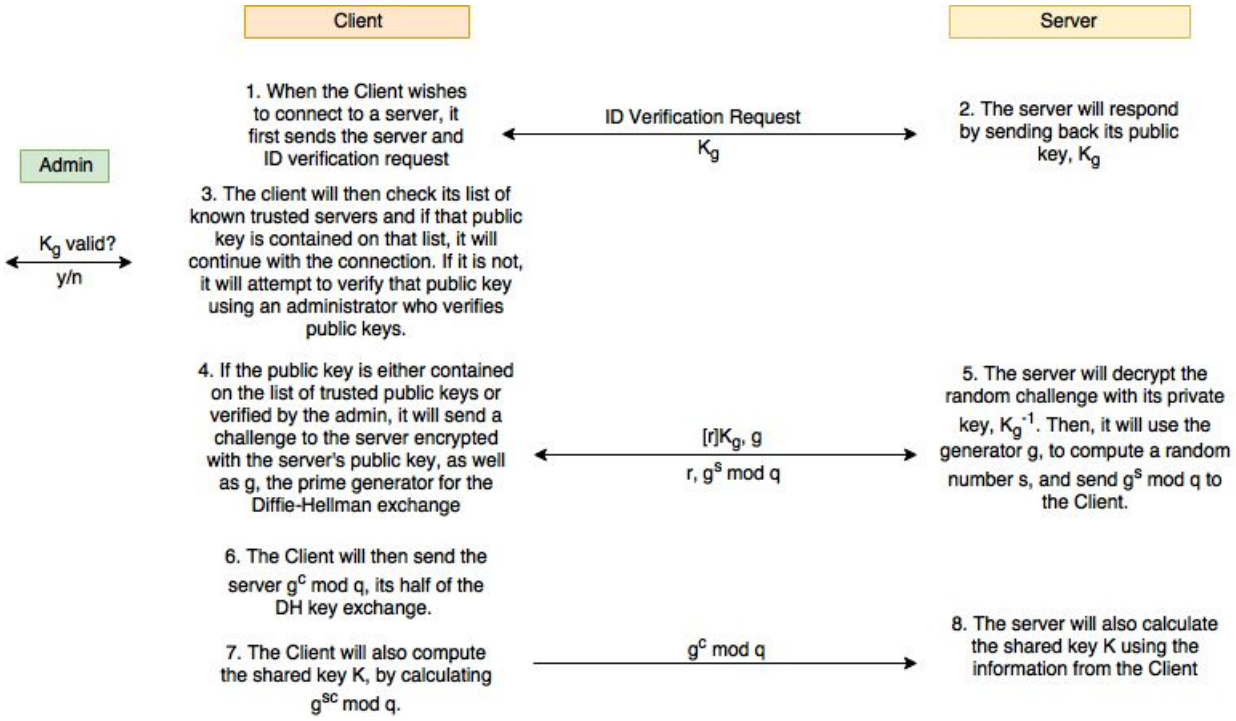
**Mechanism**

To protect against the threat of malicious file servers, our system must ensure that if a user attempts to contact a file server *s*, that they actually connect to *s* and not some other server *s'*. To accomplish this goal, our system will utilize public key authentication to verify the identity of the server to the user.

An initial connection from a user to a file server will proceed as follows:

1) The user contacts the server with an identity verification request
2) The file server then generates a key pair using 2048-bit RSA
3) The file server locally stores its private key
4) The file server then sends its public key back to the user
5) The user is presented with a message containing the file server's public key and informed that he should verify that the key belongs to the server to which he is attempting to connect. The user will be prompted to choose whether or not he wishes to continue with the connection
6) The user verifies the file server's public key through an outside channel
7) If the key is successfully verified, then the user will choose to connect, otherwise he will choose to abort the connection
8) The user will then use the server's public key to encrypt a sufficiently random challenge
9) The server will then decrypt the challenge using its private key
10) The server will then send the decrypted message back to the user
11) If the message matches the user's original challenge, then the server is considered to be authenticated, otherwise the user should close the connection to the server, since it cannot be authenticated

12) Once the files server has been authenticated, the user and the FileServer will exchange a shared key using the procedure described in T1




**Correctness**

Our mechanism for addressing the threat of Unauthorized File Servers sufficiently mitigates the threat with a few assumptions:

- No entity outside the file server has access to its private key
- Upon initial connection, the user will verify the server's host key by contacting a system administrator through an outside channel
- The user-generated challenge will be sufficiently random/unpredictable and will never be reused

The server can safely send its public key to the user over an insecure public channel because it is of little use to an attacker without the server's private key. The user can safely send the encrypted challenge to the file server over an insecure public channel without the worry of a replay attack, since an attacker cannot decrypt the challenge without the server's private key. The server can safely send the decrypted message over the same insecure public channel over which it sent its public key because of preimage resistance. If the decrypted message received by the user from the server matches the original challenge generated by the user, then the

server has been authenticated, since only the server has access to its private key, so only it can decrypt the challenge encrypted with its public key.

## T4: Information Leakage via Passive Monitoring

**Threat Description**

Our trust model assumes the existence of passive monitoring, and thus all communications need to be hidden from outside observers. Outside observers could discover the contents being communicated during a session, and potentially impersonate participants in the system. Encrypting the contents of all communications using AES-GCM for a 128-bit shared session key (exchanged already in T1), ensures that the communications remain private from any observers on the network.

**Mechanism**

Every communication will be encrypted with an AES session key exchanged with the user in T1. When passing information from the client, the client will encrypt the information using the shared key, K, and then the server will decrypt that content, with K. This ensures that any information transferred over the network can only be read with the session key K.

There are two channels of communication in the system: between the GroupServer and the Client and between the FileServer and the client. For each channel, a shared key will be exchanged using the procedure from T1, after the authenticity of both the server and the client has been established. These keys are established per-session which ensures that they cannot be reused, but also that if a key is compromised, only the information from one individual session will be compromised rather than all communications. Using 128-bit AES-GCM ensures both security and efficiency since AES is both secure and fast, as well as correctness.

**Correctness**

Using the shared symmetric key to encrypt sessions is correct under the assumption that the private key is not intercepted, and that the initial exchange in T1 is successful in that no public/private keys have been compromised. Connecting and authenticating as a user is only as secure as that user's password, so if the user has a strong password, then it will be nearly impossible for someone to impersonate that user by guessing their password to gain access to the system.

## Conclusion

Throughout the process of designing our system, we attempted to keep the design to be as straightforward as possible. We discussed the performance tradeoffs of hashing certain values, as well as using the GroupServer as a central authentication point, where the GroupServer would check every Token and every access for legitimacy. We ended up selecting things like signing the Tokens, the Diffie-Hellman key exchanges, and hashing for passwords as it keeps the design simple, but secure. We used complete mediation to ensure that every access is checked, to ensure there are no weak points for one to gain access to unauthorized actions, and we used per-session symmetric keys to ensure that if a key is compromised, only the details of one specific session would be compromised. Efficiency and thoroughness guided our design, and we think that our system succeeds in providing robust security using a minimal design to ensure that it is not only quick, but that there are fewer points of failure.

## Sources


1. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice" (PDF). Retrieved 30 October 2015.
2. "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.
3. “Has the RSA algorithm been compromised as a result of Bernstein's Paper? What key size should I be using?” https://www.rsa.com/en-us