Introduction:

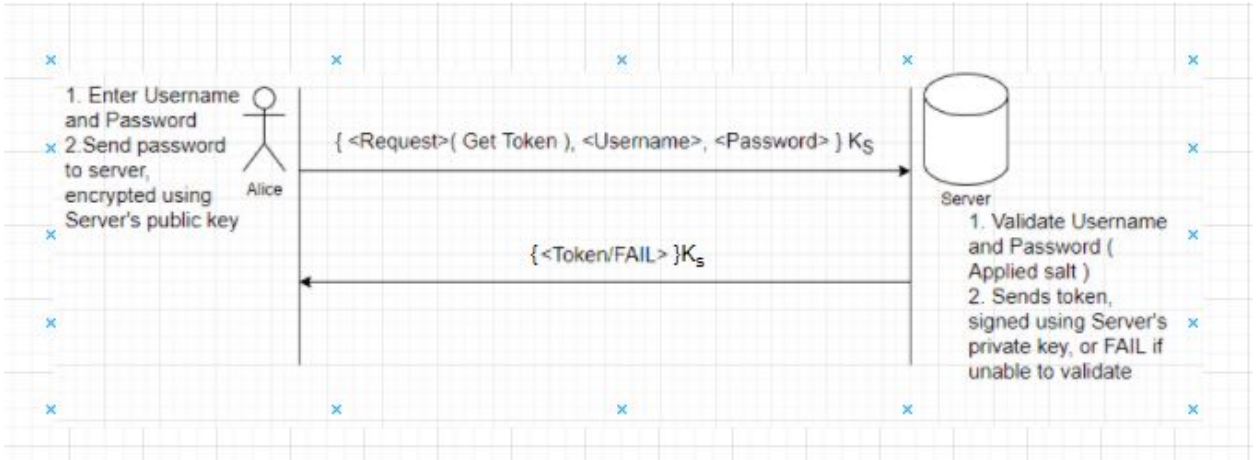
For this project, we will make use of various cryptographic techniques including symmetric and public key cryptography (128-bit AES in CTR mode and 1024-bit RSA), signatures, and Diffie-Hellman. We will use Diffie-Hellman to do session-specific key generation of symmetric keys without needing to send compromising data over the network. We will use these keys in symmetric key crypto to encrypt traffic that goes over the network. The RSA public keys will be tied to servers and help to both protect the user and verify the identity of the servers at various points in the system including encrypting information with a server's public key and signing tokens with a server's private key.

For the usage of public key cryptography, we assume that all of these keys are distributed out of band. The alternative would be for us as a group to either build our own key distribution server (which increases the mechanism of this system and leaves more opportunity for us to make an error that allows our system to be compromised) or to pay to have our public keys hosted on an existing key distribution server. It is our viewpoint that the users are already required to do some distribution of information out of band (i.e. sharing the host and port of the servers) and so adding the sharing of public keys to that out of band work seems trivial. In a real production environment, this would be susceptible to tampering and should never be done, but in a production environment we would also have our keys hosted on an existing and trusted key distribution server.

T1: Unauthorized Token Issuance

This threat may be seen being used by an adversary in an attempt to gain access to certain permissions that their account has not been granted. For example, secure, important files may be stored in File Server *F*, however an adversary does not have access to this file server. Because of this, the adversary plans to impersonate User *U* who does have access to *F* in order to view the secure, important files. Without any security policies in place, the attacker need only know the name of User *U* in order to impersonate them and gain access to files they are not supposed to have permission to view.

In order to mitigate the threat of unauthorized tokens being issued, we plan to implement a password system protocol to authenticate users before a token is issued to them. Shown in the following diagram:



Upon user creation, the user will enter a password to be stored with their account. This

password will be encrypted with the group server's public key, then sent to the group server. The server will salt the password and store a 100x bcrypt hash of the password to disk.

Upon user login attempt, the user will enter the password linked with their account. The Password entered will be sent to the group server for authentication, encrypted using the group server's public key. The server will then apply a salt to this password and then compare the 100x hash of it with what it has stored on disk. In the event of a match, the server will send the user the token signed using its private key. Otherwise the Server will send back a failed request message.

By implementing user specific passwords that only each user knows, an adversary attempting to login as a specific user other than themselves will be unable to do so. Since they do not have the required password, the server will be unable to authenticate the adversary and thus will not provide them with a token that does not belong to them.

By encrypting the messages with the Server's public key and signing with the server's private key, any passive eavesdropper will not be able to view the contents of the messages being sent.

By adding a salt to the passwords and hashing them 100 times using bcrypt, we sacrifice a slight amount of speed for a high amount of security in mitigating the threat of a brute force attack on a stolen password database. We chose to use bcrypt since it is one of the most commonly used hash functions for passwords, and is highly trusted by many.

T2: Token Modification/Forgery

The threat of token modification is one of the largest threats to this system. If any arbitrary user is able to simply modify their token as they like, then they can grant themselves access to any group's files at will. Within the token, the group access information is stored as a list so, in our naive implementation from phase 2, they are able to just append a new group name as a String to the list and they have surpassed all of the permissioning that prevented users from joining arbitrary groups and allowed only group owners to add other users to their groups.

Before we discuss how to mitigate this threat, we should discuss the assumptions we made for this policy to work:

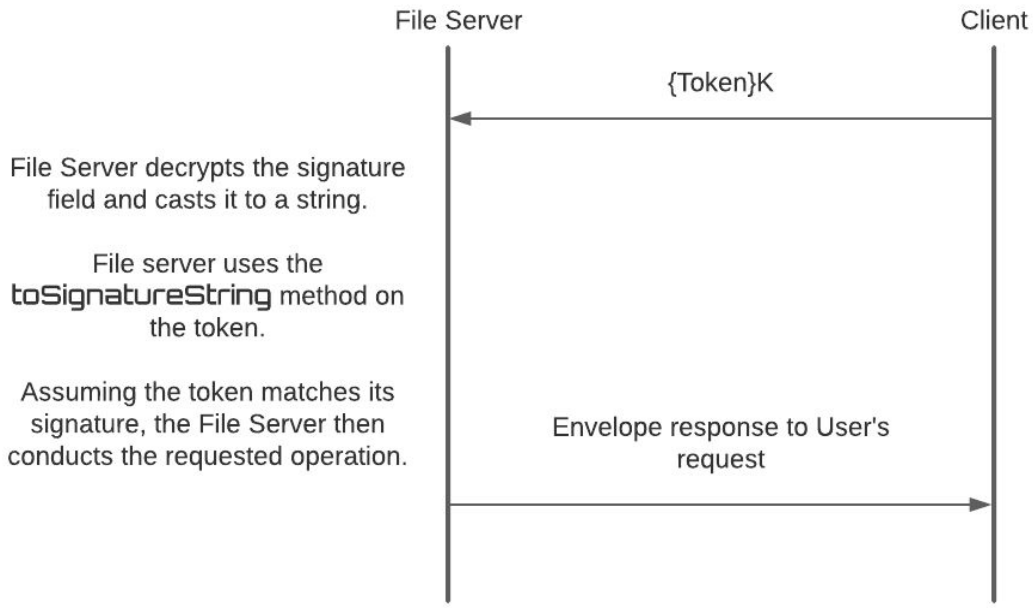
- The Group Server's public key is generated when the group server is created and is distributed out of band.
- The File Server demands a public key on startup and this public key becomes the File Server's only link to the Group Server that it trusts. From then on, it will not trust any tokens signed by any other public keys.

With these assumptions, we can mitigate the threat of token tampering by adding a signature field to the token. Once the Group Server generates the token for the user, they call a special method on it, `toSignatureString`, that formats the details of the token deterministically. The server then signs this string with their RSA private key and saves it to the signature field of the token (which is a bytes array). When the token arrives at a File Server, the file server calls the same method, then decrypts the signature and compares the two. If they match, then the token is untampered with. If they don't match, then the File Server rejects the token and sends a FAIL response to the user.

The `toSignatureString` method will generate a string that is in the JSON format. In order to make this process consistent, the resulting JSON will look something like this: `{"issuer": "alpha", "subject": "user1", "groups": ["group1", "group2", "the \"best\" group"]}`. This JSON string has the groups listed in alphabetical order and as you can see from *the "best" group*, any existing double quotes in the name of a group are escaped with a backslash. This escaping prevents users from maliciously creating a group named something like *group2", "null* which would make their group list look like `["group2", "null"]` in the signature and allow them to alter their token's group list to match this falsified signature to give them access to group2 even if they weren't in that group. With the added signature in this format, any changes that are made to the token by the user are easily detected and the token is deemed invalid.

One important note for this process is that the user's token *is* their identity with respect to the File Server, so it should be kept secret. The signing of the token is not a protection on this token from eavesdroppers, that is covered in T4 later in this paper.

The diagram below is a simple illustration of just how the process of sending tokens to a File Server would be conducted. The first step is to verify the token via the below process.



In this diagram, the encryption key K is the symmetric key generated by the D-H implementation we use that is discussed in greater detail in T4.

T3: Unauthorized File Servers

If a user U1 wishes to connect to File Server FS1, it is possible that another machine in the network, which we will call Mallory, could intercept that connection. Mallory could simply eavesdrop on the communications before sending them to the file server and do the same with communications with the file server. Worse than that, Mallory could send different files to the file

server, or fail to send the file. They could also change files that U1 tries to download or make them think there aren't files. While we cannot make them properly send network packets from U1 to FS1, we can make it so that they cannot get any information from the packets being sent, and failure to relay the packets will just cause the packets to go down a different route in the network if possible.

Our TLS-like socket encryption will make sure that once the session (the socket connection) is established, others will not be able to know the data that is being sent. So as to not repeat information, see the explanation for T4 for why this is the case. To ensure the connection is being established with the correct server, we encrypt the Diffie-Hellman key exchange being sent to the file server with the correct file server (FS1's) private key.

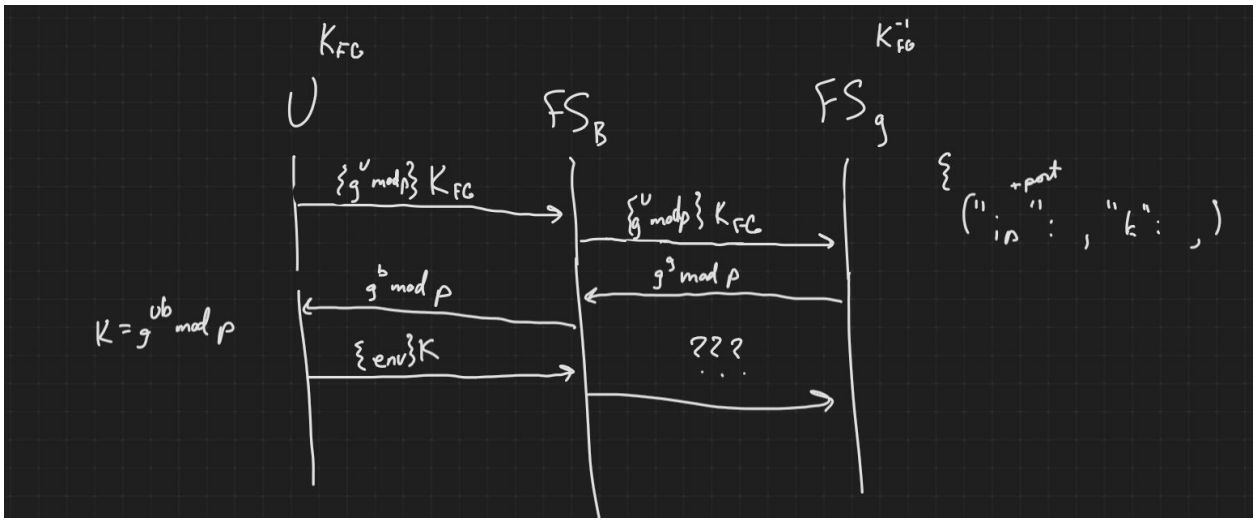
This will ensure the server we are communicating with must be correct, as only FS1 would be able to know the symmetric key. Anything encrypted with a different key would decrypt into random data. This would not be cast into the correct message type with valid strings and file objects. There is a very small chance of this, but it is negligible. See T4 diagram.

T4: Information Leakage via Passive Monitoring

If a nosy administrator (NA) can see any of the traffic, then they can use this maliciously. Not only is any privacy gone, as any of the files being sent over the network can be seen, but the NA can also obtain the ability to act as any member on the system. Even if passwords are encrypted before sending, the NA could grab the token off of the network and use that to directly communicate with the file servers as if they were the person who obtained that token from the group server. Even if the token sent was also encrypted, the NA could grab the encrypted token and use that as the file server would still decrypt it unless the encryptions keys were changed every time.

To mitigate this threat, we will directly address the ability of the NA to see the communication on the network, and we provide a changing key for the encrypted network traffic. We will do this by providing functionality like Transport Layer Security, but without authentication (which is done after this), to our network communications. To implement this, we will use symmetric encryption (AES) to encrypt data before sending it through the socket and decrypting after reading it from a socket. To generate the key, we will use Diffie-Hellman as this does not reveal any sensitive data to the network. Once the key exchange is done, both the client and server will know a shared key that they will use for only that session. We will encrypt the key we send over the network to the file server with the file server's public key to help mitigate against T3. We assume this key to be shared out-of-band. This is a fair assumption because we must already do this communication to share the group server's ip/port, so at that stage, we also give the public key for the group server.

This will make all of the network communications look like random bits and the NA will not know anything about data being sent. Even if they were able to figure out which part of the data was the token, then it would not do them any good. They could not send the encrypted token to a file server as the file server would not decrypt it to the correct thing outside of the correct socket session.



Diffie-Hellman Key Generation with Encryption