Project: Phase 3

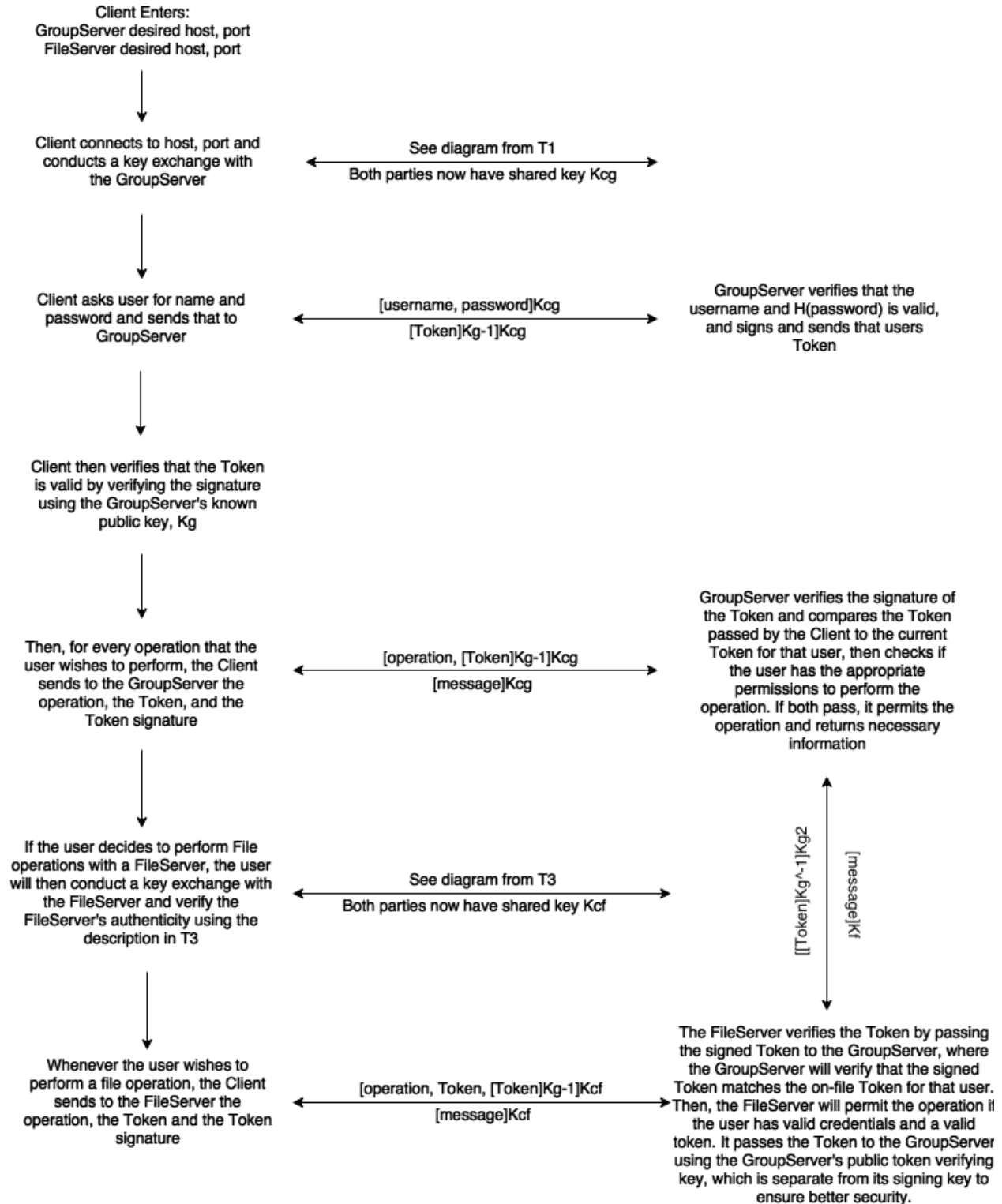Conor Lamb, Nick Mullen, Riley Marzka

Introduction

To protect against the threats detailed in this phase of the project, we chose to apply several ideas to securing our system. First, in order to secure all communications from passive attackers and those listening to network communications, all communication between servers and clients will be encrypted using 128-bit AES keys, which is a secure and fast symmetric CBC-cipher. Per-session symmetric keys will be exchanged between each Client and Server on each connection using a Diffie-Hellman exchange which gives us both a confidential initial key exchange, as well as a key exchange per session to protect against replay attacks if that session key were ever compromised.

Every access to every object is checked for permissions via Tokens. Tokens will only be issued to authenticated clients, who authenticate themselves with the GroupServer with a password, and Tokens will be signed using the GroupServer's private key. Both GroupServer and any FileServer will require both the Token signature and the Token to be passed to it in order to verify the signature and confirm that the Tokens have not been modified since being issued by the GroupServer.

Our system protects against the four threat models by establishing connections with 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.

Client Enters:
GroupServer desired host, port
FileServer desired host, port

↓

Client connects to host, port and conducts a key exchange with the GroupServer

See diagram from T1
Both parties now have shared key Kcg

↓

Client asks user for name and password and sends that to GroupServer

[username, password]Kcg
[Token]Kg-1]Kcg

GroupServer verifies that the username and H(password) is valid, and signs and sends that users Token

↓

Client then verifies that the Token is valid by verifying the signature using the GroupServer's known public key, Kg

↓

Then, for every operation that the user wishes to perform, the Client sends to the GroupServer the operation, the Token, and the Token signature

[operation, [Token]Kg-1]Kcg
[message]Kcg

GroupServer verifies the signature of the Token and compares the Token passed by the Client to the current Token for that user, then checks if the user has the appropriate permissions to perform the operation. If both pass, it permits the operation and returns necessary information

↓

If the user decides to perform File operations with a FileServer, the user will then conduct a key exchange with the FileServer and verify the FileServer's authenticity using the description in T3

See diagram from T3
Both parties now have shared key Kcf

[[Token]Kg^-1]Kg2

[message]Kf

The FileServer verifies the Token by passing the signed Token to the GroupServer, where the GroupServer will verify that the signed Token matches the on-file Token for that user. Then, the FileServer will permit the operation if the user has valid credentials and a valid token. It passes the Token to the GroupServer using the GroupServer's public token verifying key, which is separate from its signing key to ensure better security.

↓

Whenever the user wishes to perform a file operation, the Client sends to the FileServer the operation, the Token and the Token signature

[operation, Token, [Token]Kg-1]Kcf
[message]Kcf

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 current 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. 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. The token will be signed with the server's public key.

### 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.

g = agreed upon generator

q = agreed upon prime modulus

b = client's secret random number

s = server's secret random number

K = 128 bit *AES* key

Client

Server

$[g^b \bmod q]k_B^{-1}$

$[g^s \bmod q]k_S^{-1}$

$K = g^{bs} \bmod q$

$K = g^{sb} \bmod q$
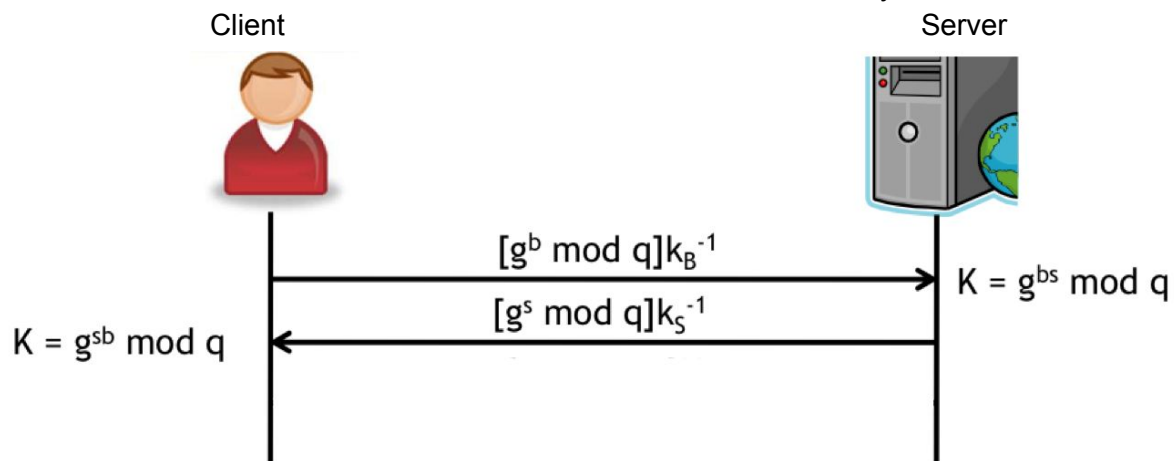
Utilizing this shared secret session key, both the client and the server will now use *AES* with and HMAC (through the GCM mode) to start the authentication process over an encrypted channel. AES is a highly secure CBC-cipher which allows symmetric keys to encrypt communication between two parties. HMAC is a mac address generated with a shared hash function and the contents of the message, which, when appended to the confidential AES ciphertext, secures the integrity of the transmitted message between the two parties. This will preserve both the confidentiality and integrity of the authentication process.
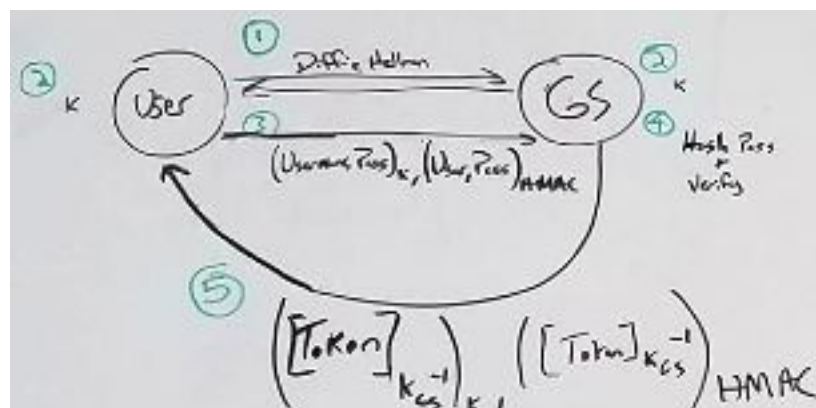
Solution Reasoning

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 our 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 and backed up integrity-wise by the HMAC. In other words, communication between the server and client will appear completely random to adversaries and if they try to augment it, the server will know. The same goes for the return trip of the token to the user.

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.
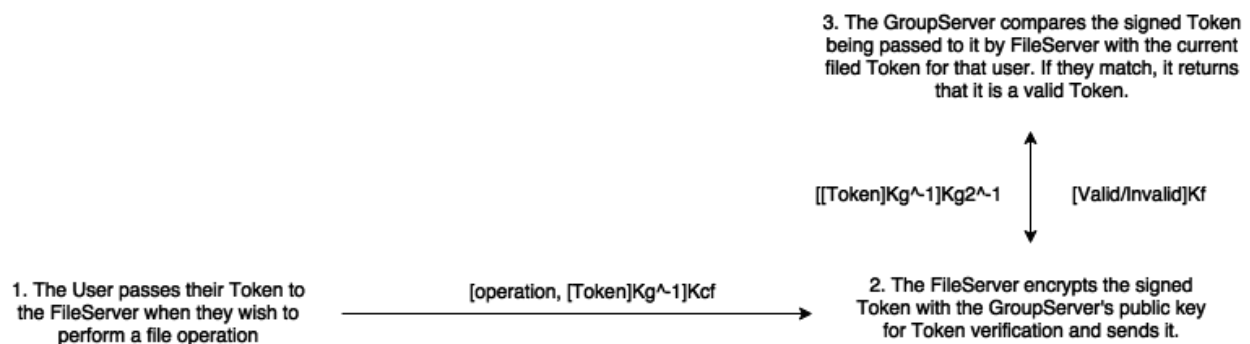
Mechanism

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, which has been signed with the GroupServer's private key.

The client can then verify the authenticity of the token by verifying the signature with the GroupServer's public key, which is a 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. Note that the GroupServer has 2 RSA key pairs. One is intended to sign tokens, the other is intended to communicate with FileServers. To verify that the token is current, the FileServer will send the signed token, encrypted using the GroupServer's second public key to the GroupServer. The GroupServer will then hash the token received from the FileServer and compare it to a hash of the current user token which the GroupServer has stored. The GroupServer will respond to the FileServer to say whether the token is valid. Its response will be encrypted using the FileServer's public key (discussed in T3).
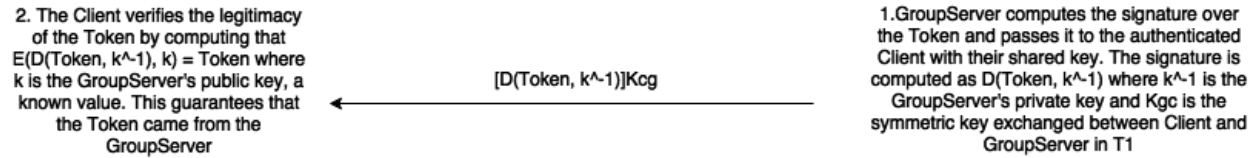
The public/private keypair for the GroupServer is 2048-bit RSA which is sufficient for security against brute-force attacks, and efficient enough to complete operations quickly. The key exchange for the symmetric key and the symmetric key itself is described in further detail in T1.

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.

3. The GroupServer compares the signed Token being passed to it by FileServer with the current filed Token for that user. If they match, it returns that it is a valid Token.

$[[Token]Kg^{-1}]Kg2^{-1}$ $[Valid/Invalid]Kf$

1. The User passes their Token to the FileServer when they wish to perform a file operation

$[operation, [Token]Kg^{-1}]Kcf$

2. The FileServer encrypts the signed Token with the GroupServer's public key for Token verification and sends it.

Issuance of a Token

1. GroupServer computes a signature of the token with its private key, K^{-1}, and passes the signed token to the user by encrypting it with the shared symmetric key
2. The user can verify the signature using the GroupServer's public key, K.

2. The Client verifies the legitimacy of the Token by computing that $E(D(Token, k^{-1}), k) = Token$ where k is the GroupServer's public key, a known value. This guarantees that the Token came from the GroupServer

$[D(Token, k^{-1})]Kcg$

1.GroupServer computes the signature over the Token and passes it to the authenticated Client with their shared key. The signature is computed as $D(Token, k^{-1})$ where k^{-1} is the GroupServer's private key and Kgc is the symmetric key exchanged between Client and GroupServer in T1

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

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 Token object, the server will reject the operation, ensuring that forgery of a Token will not allow privileges to be modified. As for validation of the signed token by the GroupServer, by sending a signed token encrypted with the GroupServer's second public key, we prevent an attacker from brute-force generating tokens and sending them to the GroupServer until he finds one that is valid.

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
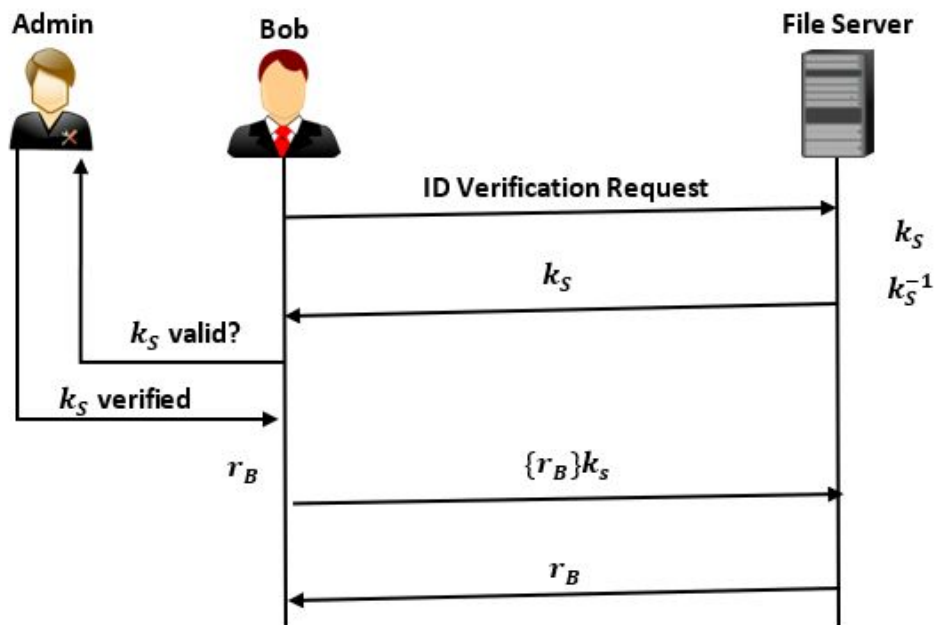
Correctness

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 and 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 in AES using the 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 ensures both security and efficiency since AES is both secure and fast.

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.