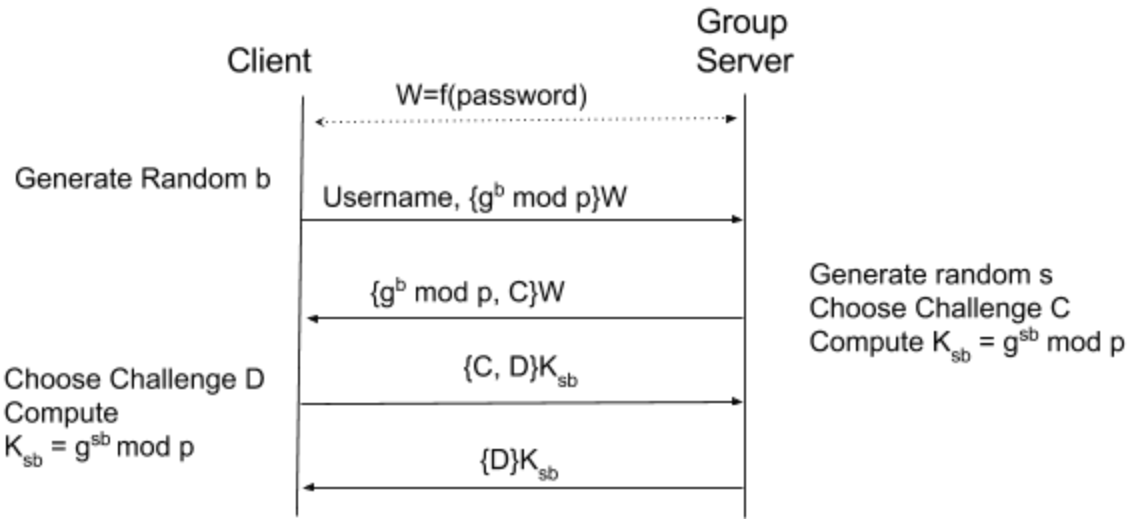
Introduction

Our group used various cryptographic measures to handle each of the treat models. For token issuance we used Encrypted Key Exchange to create a strong key from a weak password entered by the user. Then both the group server and the client will authenticate each other using AES encryption with the key generated from EKE. To handle token modification, we used HMAC-SHA2 to verify that the token isn't going through any unwanted modification. To handle unauthorized file servers, we had the file server send over a fingerprint to the client. The client is then expected to verify if that fingerprint matches the fingerprint that the admin has saved. Finally to handle the passive attacker we encrypt all messages between client and server. We are encrypting everything using AES-128 because in the group server we used EKE to authenticate and generate a symmetric key. This key will be used for the AES encryption. The file server uses Diffie Hellman to generate a shared key that is then used for the AES encryption.

T1 Unauthorized Token Issuance

The issue describes unauthorized clients accessing data they shouldn't be able to see. The best way to prevent this kind of attack is to make sure that every client is an authorized member of the system. To ensure that each member currently using the system is authorized, we just have to make sure they only have access once they provide a password. This will be done prior to any other methods and the user will only be able to use the system when they prove they have the correct password. A clean way to make sure that the password is hidden from passive attackers is to encrypt the messages from both the server and the client. The protocol that will be used is Encrypted Key Exchange. This protocol ensures that a weak password will yield a strong encryption key that will protect all data from passive attackers. The W generated from the password will originally be a symmetric key which will be used for AES encryption in the EKE protocol. EKE was chosen because it provides a shared key and authenticates the user at the same time. EKE was chosen over SRP because we are assume that the group server is trustworthy meaning we can assume it will not be compromised. With this assumption in place, we won't need to protect against compromised servers. The communication between client-server authentication is done through AES-128 using the shared key generated earlier during EKE.



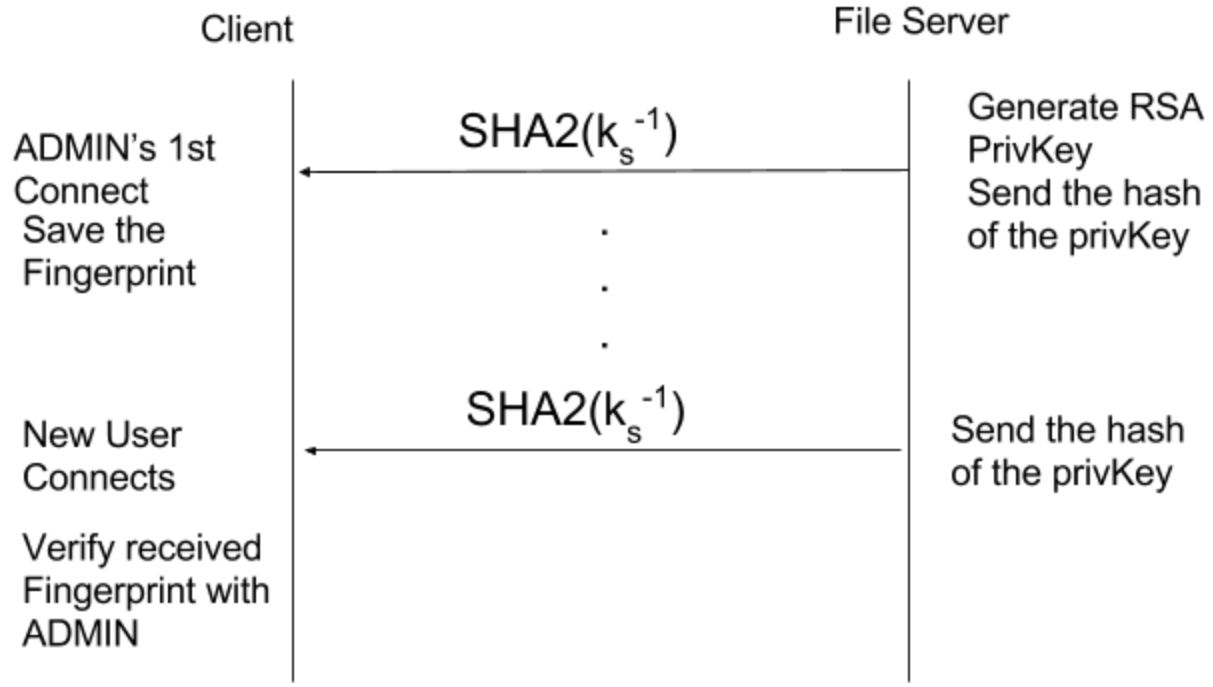
T2 Token Modification/Forgery

Token modification is taken as a client attempting to modify the group list within their token in order to gain access to more groups. Our current implementation doesn't use token class to check for ownership or admin so that's not the concern of this problem. In order to verify that the token isn't changed we create a method in the UserToken interface where we generate a hash value and store the hash value in a private variable. This generate hash method is called after each user modifications (addusertogroup, deleteuserfromgroup, etc.) to make sure that the hash value is up-to-date. When ready to check we send the userToken and the hash value to the fileserver. The server then generates its own hash of the token sent and compares that hash to the sent hash. If these two values do not equal then the token has been modified and should not be accepted. First of all hashing is used because of its property whereby a small change in the object usually has a large change in the hash value. Therefore, if the client was able to modify the token, this modification will noticeably change the hash generated. The hash function we will use is HMAC-SHA2 because it gives us the benefits of preimage, second preimage, collision resistance and protects against length extension attacks.

T3 Unauthorized File Servers

This problem focuses on the client being trustworthy and testing the server to see if it's also trustworthy. In other word we have a trusted client trying to see if the server can be trusted or not. Initially when a file server is up and running for the first time, a fingerprint will be generated. In the project implementation, the fingerprint would be printed on the terminal running the file server. Every user is assumed to have the fingerprint(s) because they can ask their boss or check the fingerprints copied down somewhere in the work building. The file server

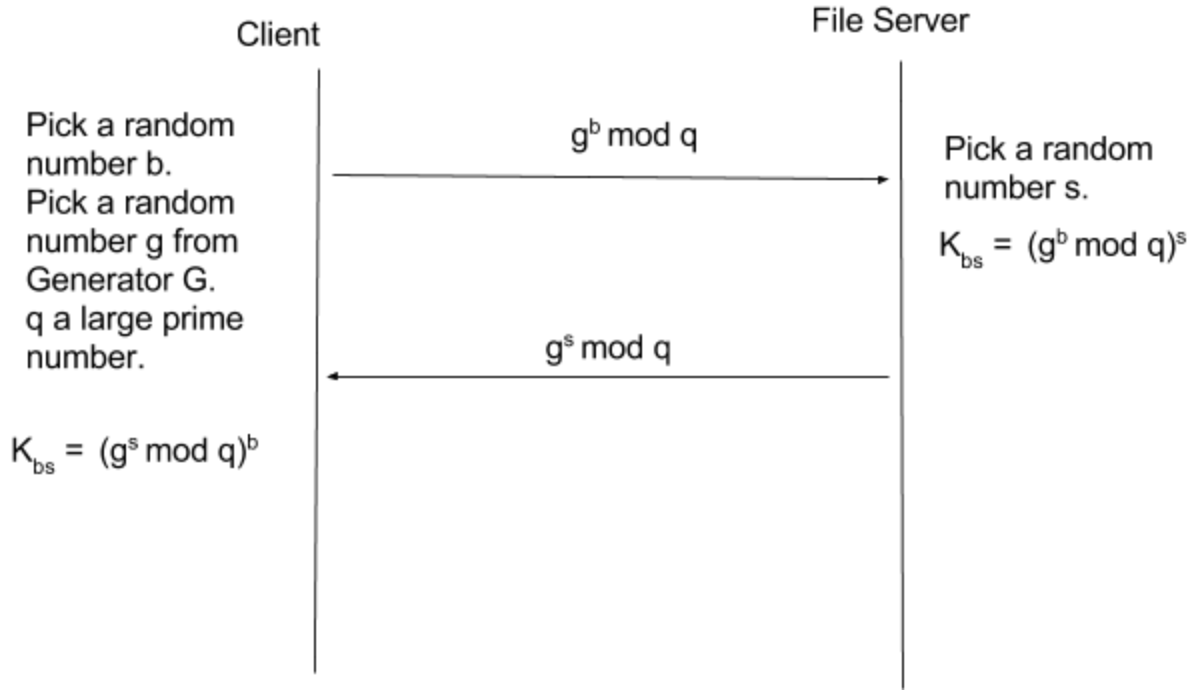
will send the fingerprint over to the client when the client connects. The client is to verify if the received fingerprint matches the saved fingerprint. To reiterate, the client is assumed to not be lazy with the authentication. The fingerprint will be a hash of an RSA private key of length 128 bits. The hash function that will be used is SHA2 because any hashing algorithm will work but SHA2 is picked for its quick performance. We are using an RSA private key because if a public key was used then anyone can just look up the server's public resulting in any malicious intent by just copying the public key. RSA key is used because SSH commonly uses an RSA key gen. Finally 128 bits because that is a commonly used fingerprint size for SSH. Too large of a bit size could complicate storing for the admin and could result in a lot of traffic. Too small and we risk replay attacks by attackers brute forcing the fingerprint.



T4: Information Leakage via Passive Monitoring

The problem is that passive monitoring allows for attackers to see messages between client and servers. In order to prevent this, we make sure messages are encrypted when sent through transit, then decrypted when the recipient receives the message. We will use a single encryption scheme (AES) for both Group and File servers. For the Group server, we use the Encrypted Key Exchange protocol to create a shared, symmetric key between the client and server since access to the Group server requires a password, which can potentially be weak. EKE allows us to generate a strong key out of that password. For the File server, we use Diffie Hellman to

generate a shared key between the client and server instead of EKE because the File server doesn't require/have access to the client's password. All passwords are stored and restricted to the Group server, and the File server cannot directly communicate with the Group server. Since both the Group and File servers generate a shared key, the best encryption algorithm that makes sense to use is a symmetric crypto algorithm. The reason for AES over other symmetric algorithms is because currently AES is one of the strongest, recommended crypto symmetric key algorithms to date. Specifically we will use AES-128 since AES-256 may be overkill in our context. As mentioned above, we briefly use RSA to authenticate the client and File server, and the rest of the communication will be encrypted in AES-128.



Conclusion

In conclusion, our system used various algorithm to implement a strong file sharing. The main flow between each mechanism is that the protocols used for authentication generated a strong key which can be used for encryption. In the group server we used a protocol that generated a strong, shared key from a password required in T1 and we used that key in the AES encryption for T4. Initially to authenticate the server to the client we tried to use the SSH key protocol. Our group created a simplified version of the SSH key protocol but it failed. The reason it failed was because we didn't develop a secret that the client and the requested server shared to correctly authenticate the requested server. In other words, a malicious server posing as the file server doesn't actually have proof that it's authentic. We ended up misunderstanding the SSH key

protocol and forgot about the fingerprinting aspect of it. That is when we incorporated fingerprinting to do remote host authentication. Our version of fingerprinting basically replicated the process of how it would work in an SSH connection by hashing a key (in our case, a private RSA key) and taking advantage of its preimage resistance.

References

https://security.stackexchange.com/questions/1751/what-are-the-realistic-and-most-secure-crypto-for-symmetric-asymmetric-hash
https://docs.bmc.com/docs/display/itda27/About+the+SSH+host+key+fingerprint
https://superuser.com/questions/421997/what-is-a-ssh-key-fingerprint-and-how-is-it-generated
http://www.lysium.de/blog/index.php?/archives/186-How-to-get-ssh-server-fingerprint-information.html