Introduction

Our group used various cryptographic measures to handle each of the threat models. For token issuance we use Secure Remote Protocol (SRP) to create a strong key from a weak password entered by the user. After the group server has significant assurance that the client is actually who they say they are, it will send a signed UserToken encrypted with a shared key. The signature is to ensure data integrity and the encryption is to ensure confidentiality. To handle unauthorized file servers, we used Authenticated Diffie Hellman. This combines public key crypto and Diffie Hellman to authenticate both sides and create a strong key by the end of the protocol. Finally to handle the passive attacker, we encrypt all messages between client and server. We are encrypting everything using AES-256 because in the group server we used SRP to authenticate and generate a symmetric key. The file server uses Diffie Hellman to generate a shared key that is then used for the AES encryption. The keys used will be of length 256 and the mode of operation will be CFB because most of the work done with the file sharing system will be done with files and CFB is the best for file encryption. Any time public key crypto is used we will encrypt using RSA because it is the most commonly used asymmetric crypto and it is known for its strength in security.

T1 Unauthorized Token Issuance

https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#DH2Ex

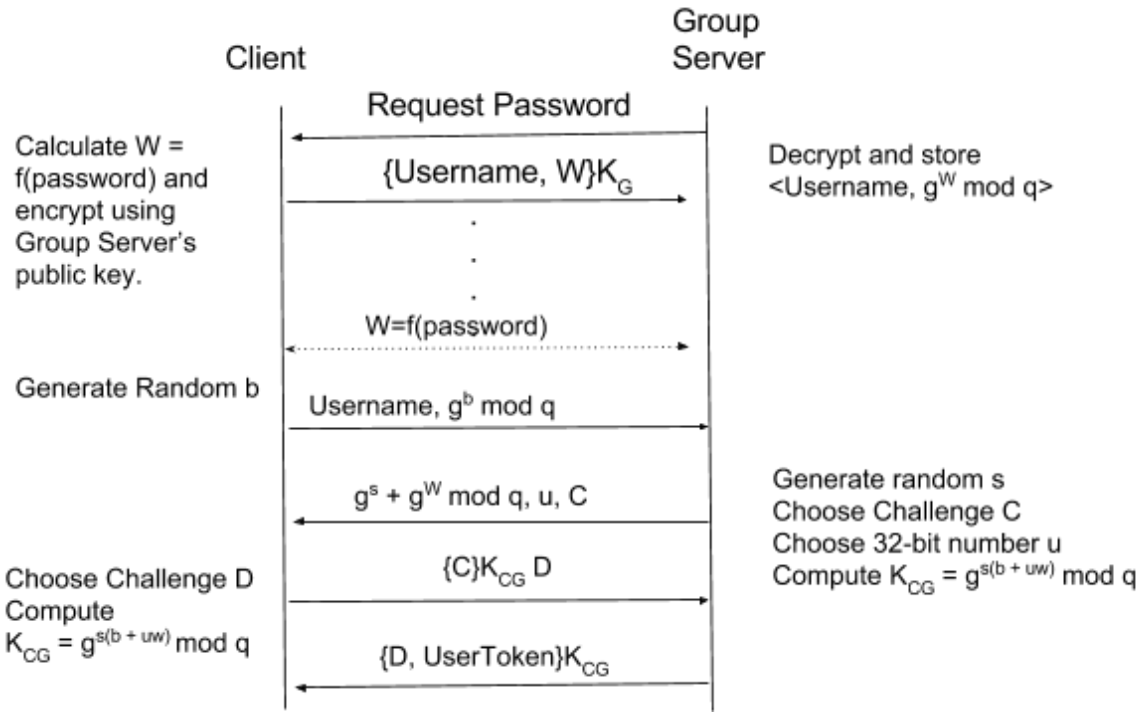
https://www.programcreek.com/java-api-examples/index.php?api=org.bouncycastle.crypto.macs.HMac - example 4

https://github.com/bcgit/bc-java/blob/master/core/src/main/java/org/bouncycastle/crypto/agreement/srp/SRP6Server.java

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 Secure Remote Protocol (SRP). This protocol ensures that a weak password will yield a strong encryption key that will protect all data from passive attackers. The Diffie Hellman group that will be used is Group 24 because we will be using size 256-bit keys and Group 24 is the recommended group by Cisco for its security. The W will be generated by hashing the password with HMAC-SHA256. For the initial setup, the server will request a password. The client will

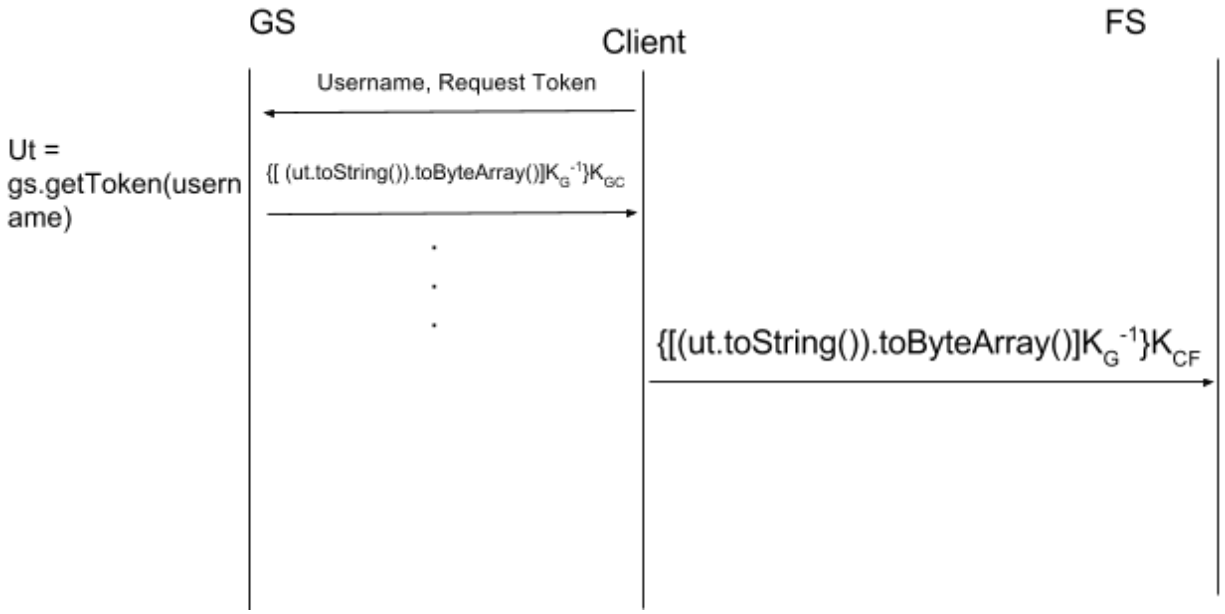
respond by encrypting W with the group's public key. The group server will decrypt and store it as a username and a modular function, g^w mod q, of w as a pair. Later on, the client will generate a random number b and send it as a g^b mod q along with their username. The group server will generate a random number s, a random 32-bit challenge C, a 32-bit number u, send C, u, and $g^s + g^w$ mod q, and finally compute the shared key $K_{CG} = g^{s(b+uw)}$ mod q with the user's generated b. The user will then compute his shared key K_{CG}. The user will generate a 32-bit challenge D. The user will then send D and his encrypted response to C with K_{CG} to the server, so the server can authenticate the user. Finally the server will encrypt his response D and send it over to the user. The size of the challenges isn't too important. As long as the size is large enough that an attack can't guess the number, the protocol will work. The size of the secret keys s, b, and w will be 256 bits because we found a source that argues that DH should be at least 256 bits to ensure that it takes 2^{128} steps to calculate the discrete log. Also because our hash returns a 256 bit value we should use a 256 bit size for all the secret to simplify the implementation. At this point, both sides have authenticated each other and have generated a shared key that only they can see.

Our decision for SRP is because it satisfies the "Big 3" requirements we found, which were: resistance against dictionary attacks, perfect forward secrecy, and non plaintext-equivalence. According to a source from Stanford University, EKE doesn't protect against all three of those requirements, while SRP does. SRP is secure because it uses the Discrete Logarithm Problem property to hide the secrets: w, b, and s. The source claims that SRP is fast, easy to implement, and an overall simple protocol, all of which is perfect for our use in order to create a quick, secure authentication process.



T2 Token Modification/Forgery

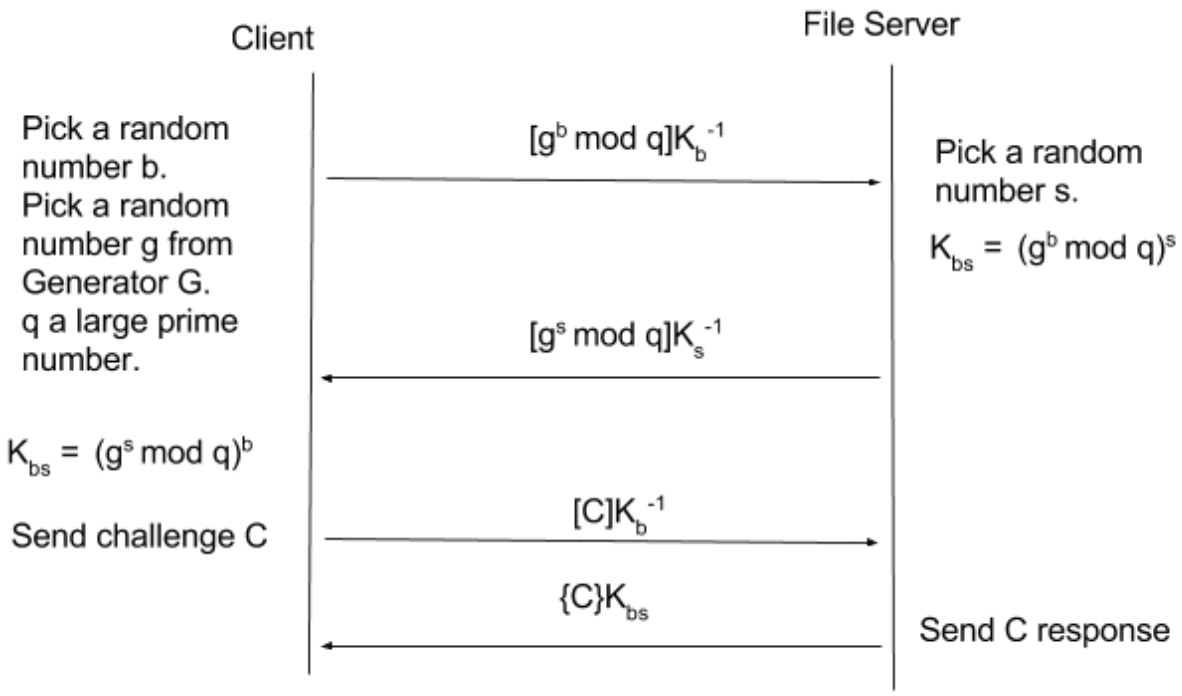
Token modification is taken as a client attempting to modify the group list within their token in order to gain access to more groups. Our current implementation doesn't use token class to check for ownership or admin so that's not the concern of this problem. Initially when the client wants to do anything, they will have to first connect to the group server to request a UserToken. The group server will stringify the UserToken and convert it into a byte array in order to sign the token to send to the client. The method of stringifying will consist of concatenating the subject (client), followed by the issuer (server), followed by the sorted list of the client's groups. We will sign the byte array using the group server's private key and encrypt it using the shared key between group server and client. This process will happen every time the group server is about to give a token to the client. When the client wants to connect to the fileserver, he just has to decrypt the encryption from the shared key of the signed UserToken, encrypt the signed token with the shared key between the client and file server, and send it to the file server. The file server will verify the signature by decrypting using the client-file server shared key, and then decrypting the signature with the group server's public key. If the group's server signature is still intact and the file server can break the signature to obtain the UserToken, it means the group server was the last one to make a change to the token, meaning the token is valid. Otherwise, it means someone broke the signature and possibly made changes to the token. Therefore the file server will stop the connection if it couldn't verify that the signature is from the group server.



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 words, we have a trusted client trying to see if the server can be trusted or not. To authenticate the server, we decided to use Authenticated Diffie Hellman. We first use public key crypto and use the private keys to sign the messages sent over. This ensures data integrity because anyone can verify or break a signature, but only the desired client or server can sign the message. In our case, the client sends a signed Diffie Hellman message. The server knows that the message is from the client because the message was signed with the client's private key. The server verifies this by decrypting the message using the client's public key. Now the server sends over a signed DH message. The client can verify that the message came from the desired server by decrypting the signature with the server's public key. After decryption, if the message doesn't make sense, then the client knows the message either came from some unknown attacker or has been modified in transit so he can ignore it. After both sides have authenticated each other, they will calculate their shared symmetric key from the Diffie Hellman protocol. Finally the client sends a random challenge to verify that the server has the same key. The server responds by encrypting the challenge with the shared key. At this point, both sides have authenticated each other and have a new symmetric key which they will use to encrypt messages for the rest of the session.

The DH group that will be used is again Group 24 because it's the commonly used group for key lengths of 256. The symmetric encryption will be AES and the public key encryption will be RSA. The challenge will be a randomly generated 32 bit number.



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. For the group server, we used SRP as described in T1 to create a symmetric key and this key will be used for AES encryption. For the file server, we used an Authenticated Diffie Hellman to generate a shared symmetric key as described in the model in T3. In both servers, AES encryption will be used with a symmetric key generated from the authentication protocol so any passive monitor will only see garbage in transit. Since both key lengths are set to be 256, the encryption will be AES-256.

Conclusion

In conclusion, we used various algorithms to implement a strong file sharing system. 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. To fix this we used public key crypto to ensure that each side had a secret. This secret was their private key. Then we used Diffie Hellman to make a symmetric key. This was done to make the encryption more efficient by using AES instead of RSA. In the end, the authentication processes gave us a method to create a key to encrypt everything after authentication.

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

http://srp.stanford.edu/advantages.html

https://supportforums.cisco.com/t5/security-documents/diffie-hellman-groups/ta-p/3147010

https://crypto.stackexchange.com/questions/1975/what-should-be-the-size-of-a-diffie-hellman-private-key