Cipher Squad Phase 4 Writeup

Members: Aidan Lowe, Zach Schultz, Jordan Shopp, Hayden Walker, Alex Zhou
Pitt ID: ail39, zjs32, jas749, hdw9, yuz210

Introduction

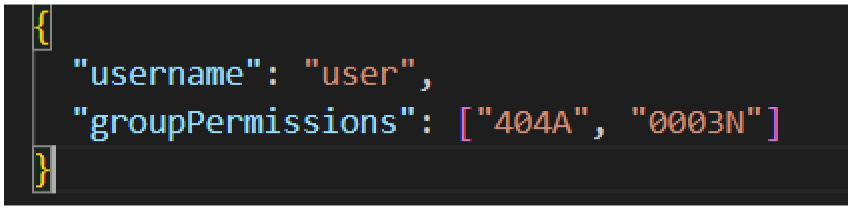
In our system's current iteration, we have successfully implemented protocols that address threats 1 through 4. Our current implementation, however, is still left vulnerable to threats 5-7. Some aspects of threats 5-7 are already addressed in our current implementation (for example, in our current implementation, an attacker would be unable to figure out how to modify ciphertext in a way that produces a desired plaintext) , but each threat requires at least some changes to our system to be addressed fully. Therefore, we have come up with methods to fully address each threat. For threat 5 (message reorder, replay, or modification), a timestamp and counter will be added to each message; the timestamp will prevent replay and the counter will make it possible to ensure that the messages are in the correct order. Additionally, a SHA-256 hash of the plaintext of each server-client communication will be added so that it can be ensured that the ciphertext of a message was not modified. To address threat 6 (private resource leakage), we plan on encrypting all messages with a key that only clients with permission to view the message (and not the resource server the messages are stored on) will have access to. Finally, we plan to address threat 7 (token theft) by adding a new attribute to user tokens; each user's token will contain the public key of the resource server that the user chooses to connect to.

Assumptions (Assumption 1-4 is from phase 3, so no explanation is restated here):

1. There exists a trusted website / application in which server IPs, port numbers, and their public keys are stored and available for users to view and access. The AS public key and IP are also stored here. To open their servers to the public, the server operator will have to provide their relevant information on the website. The server operators will update this information truthfully. Server owners will upload their respective information, and users will have the ability to browse through them. This means that the server's public key is known by the user (along with their IP and port number), so when the client connects to a server, it can automatically pull the relevant public key. The users will only have read access to the information. In the case where a server is not open to the public, the users of the servers will receive the relevant information to access the server beforehand through the server operator.
2. The authentication server and resource servers securely store and protect their respective private keys. Servers will update their public/private key pair as necessary (if it's compromised) and will reflect these updates in the trusted application for the user to see.
3. The authentication server is completely trusted.
4. There is a universally accepted and known large prime number q and generator g used for Diffie-Hellman key generation for AS communication.
5. The attacker has full read access over the communications channel between servers and clients, and are capable to perform active attacks such as server & user impersonation, manipulation of message ordering. And could be a rogue RS that could leak private information to unauthorized individuals, or attempt to steal user tokens for other users to use.
    a. Justification: Provided in Phase 4 Document

For threats that require public key cryptography, we will employ RSA SHA-256 with a 4096-bit key size and PSS padding. SHA-256 will be used as our hash algorithm due to its robustness and suitability for cryptographic protocols. A 4096-bit key size is used to prevent a brute force attack and ensure enhanced security at the cost of increased processing time. PSS padding is used as it suits well for digital signatures.

Tokens will be in the form of the json object. The token contains the username, and the group permissions of the user (something like below) For transmission, the token will be serialized into a string so it can be signed or encrypted.



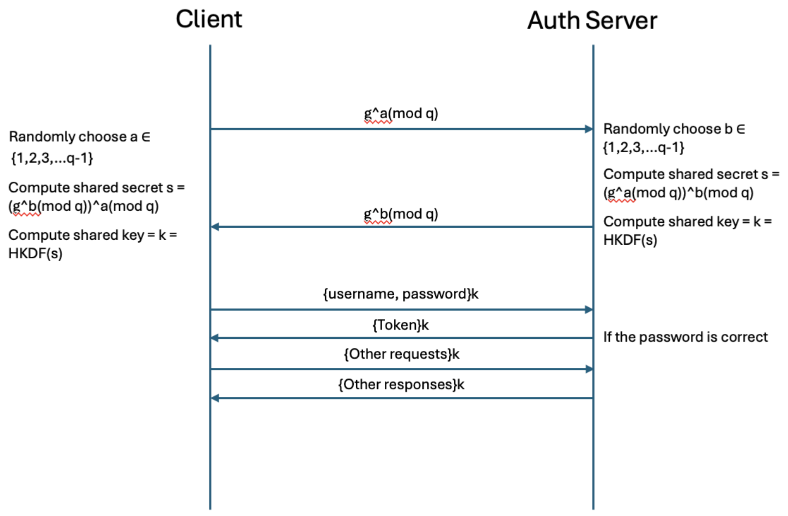
```
{
  "username": "user",
  "groupPermissions": ["404A", "0003N"]
}
```

Example Token, group permission is formatted by the group number following the permission level, A for admin, N for regular users

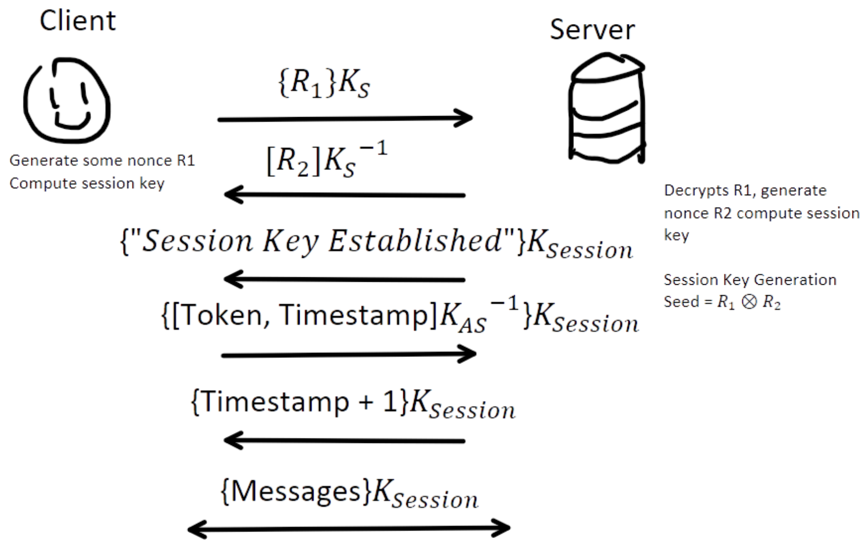
__Current Security Protocol:__

Authentication Server:

Upon user connection, the client and the AS will be using secp256r1 Diffie-Hillman elliptic curve to exchange a shared secret that will be used for AES encryption between the two parties. Both clients will then use a HMAC-based Extract-and-Expand Key Derivation Function (HKDF) to derive the 256-bit AES key. The AES key then will be used as the session key for the remaining session. The AES encryption will be done through CBC mode with 256 bit key and PKCS7 Padding. After the key exchange, the client will pass their username and password to be authenticated. Then after performing all operations they want through the AS (i.e. group permission management, etc), they would request a token to be sent by the AS so they can connect to the resource servers.


Client
Auth Server
g^a(mod q)
Randomly choose a ∈ {1,2,3,...q-1}
Compute shared secret s = (g^b(mod q))^a(mod q)
Compute shared key = k = HKDF(s)
Randomly choose b ∈ {1,2,3,...q-1}
Compute shared secret s = (g^a(mod q))^b(mod q)
Compute shared key = k = HKDF(s)
g^b(mod q)
{username, password}k
{Token}k
If the password is correct
{Other requests}k
{Other responses}k

Resource Server:
Upon connecting to the RS, the client will generate a 32 byte nonce, and send the nonce to the RS after encrypting it using RSA public key encryption with the RS' public key. The RS will generate a 32 byte nonce of their own, sign it using their private key, and send that to the client. The two nonces will be XORed together to and used as a seed to generate a 128-bit AES key, which will be used as the session key for the exchanges. The AES encryption will be using CFB mode with no padding. The server will then send a confirmation message encrypted using the session key to indicate the key exchange has been completed. Client will then send the signed token, after encrypting with session key, to the RS. Whom after processing the token, will send back a timestamp to indicate the completion of token processing. After which the two parties will continue to normal RS services such as group selection and moving to chat rooms.


Client
Server
$\{R_1\}K_S$
Generate some nonce R1
Compute session key
$[R_2]K_S^{-1}$
Decrypts R1, generate nonce R2 compute session key
Session Key Generation
Seed = $R_1 \otimes R_2$
$\{"Session\ Key\ Established"\}K_{Session}$
$\{[\text{Token, Timestamp}]K_{AS}^{-1}\}K_{Session}$
$\{\text{Timestamp} + 1\}K_{Session}$
$\{\text{Messages}\}K_{Session}$

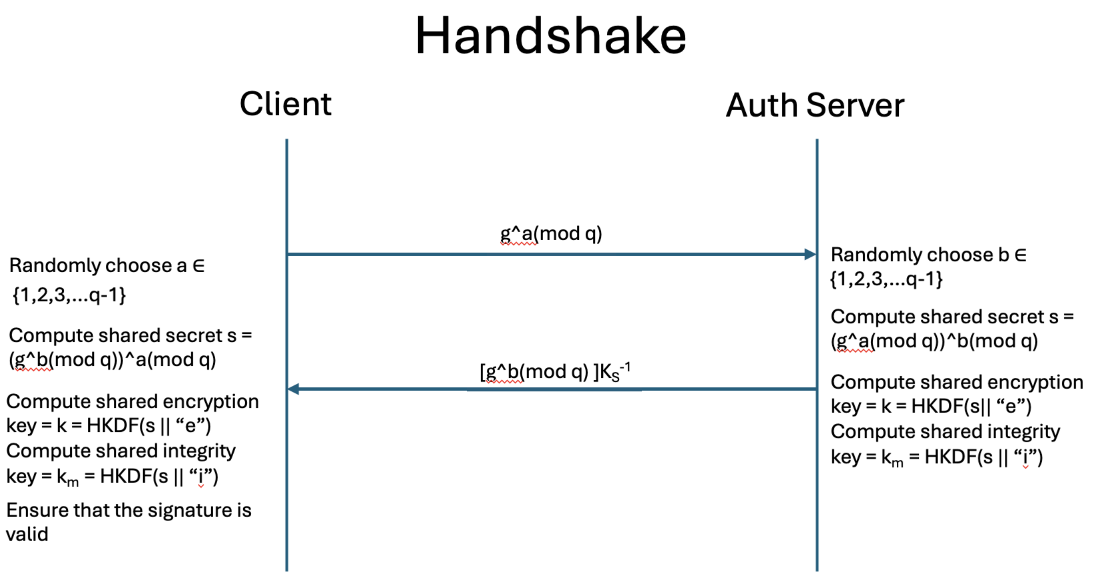
<u>Threat 5: Message reorder, replay, or modification</u>

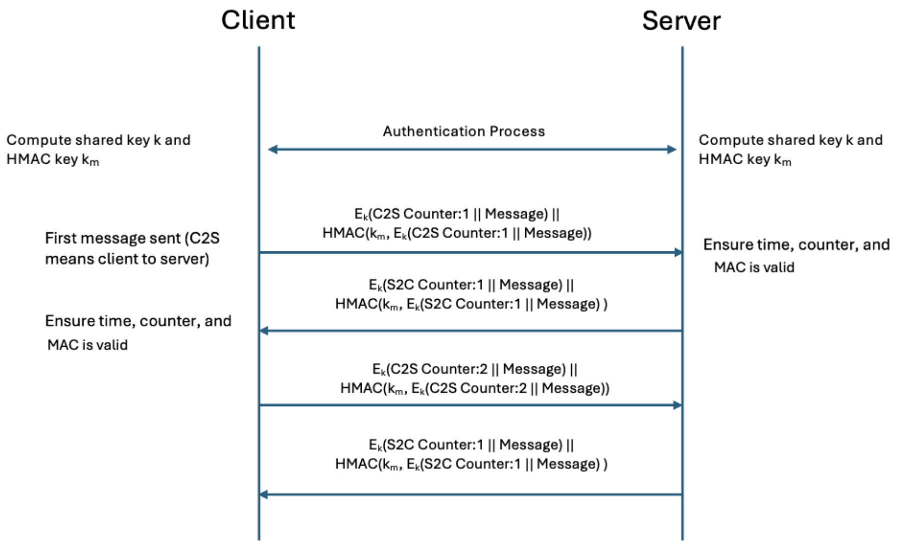
The threat covered in this section involves an active attacker being able to impersonate the user to the server by listening to the messages between the user and the server, as well as change what is sent to the server. We have modified our message protocol in order to prevent these attacks, and ensure that the messages are sent in order by the right person at the right time.

For message replay and reorder, we plan to prevent this by adding a counter (encrypted with the shared AES key) to each message with a count integer (starting at 1), that identifies the order of the message. Each direction of communication (server to client and client to server), would have their own counter to prevent concurrency issues where both actors send messages at about the same time. These counters would be checked to ensure the message is sent in order, otherwise the connection would be terminated. This would prevent an active attacker from being capable of changing the order of messages being sent. (NO TIMESTAMP).

In addition to this, the AS-client and RS-client connections each generate a random key each session. This means that only the intended interactors (AS and client or RS and client) have keys used for encrypting messages between the pairs. This, paired with the assumption that an attacker would not be able to crack 128 or 256-bit AES encryption, means that an attacker would not be able to interpret plaintext from the ciphertext sent across connections, nor be able to generate their own ciphertext from a desired plaintext. Though, they would be able to modify the ciphertext, which would possibly affect what is being sent, even if the attacker would not be able to know what effect it has on the plaintext. Thus, we want to add a SHA-256 MAC of the ciphertext of every cryptographically secure communication between the servers and the client, which would be checked to secure message validity. In order to generate the encryption key k and the HMAC key km, we are adding to our Diffie Hellman exchange algorithm by using the shared secret to generate both km and k. To do this, we are appending "e" and "i" to the shared secret as it gets processed by the HKDF, resulting in two different keys derived from the same shared secret. We plan to authenticate that the AS is indeed the AS by having it sign it's portion of the Diffie-Hellman exchange, using RSA with a 4096-bit key, SHA-256 hashing algorithm, and PSS padding and the AS private key. This comes with the added assumption that all clients would know the AS's public key, which is a reasonable assumption knowing that there is only 1 AS.

Message integrity will be checked by both the servers and the client by ensuring that the HMAC of the message is indeed accurate given the ciphertext sent.


Handshake
Client
Auth Server
g^a(mod q)
Randomly choose a ∈ {1,2,3,...q-1}
Compute shared secret s = (g^b(mod q))^a(mod q)
Compute shared encryption key = k = HKDF(s || "e")
Compute shared integrity key = k_m = HKDF(s || "i")
Ensure that the signature is valid
Randomly choose b ∈ {1,2,3,...q-1}
Compute shared secret s = (g^a(mod q))^b(mod q)
[g^b(mod q)]K_S^{-1}
Compute shared encryption key = k = HKDF(s|| "e")
Compute shared integrity key = k_m = HKDF(s || "i")


Client
Server
Compute shared key k and HMAC key k_m
Authentication Process
Compute shared key k and HMAC key k_m
E_k(C2S Counter:1 || Message) || HMAC(k_m, E_k(C2S Counter:1 || Message))
First message sent (C2S means client to server)
Ensure time, counter, and MAC is valid
E_k(S2C Counter:1 || Message) || HMAC(k_m, E_k(S2C Counter:1 || Message))
Ensure time, counter, and MAC is valid
E_k(C2S Counter:2 || Message) || HMAC(k_m, E_k(C2S Counter:2 || Message))
E_k(S2C Counter:1 || Message) || HMAC(k_m, E_k(S2C Counter:1 || Message))

Threat 6: Private Resource Leakage

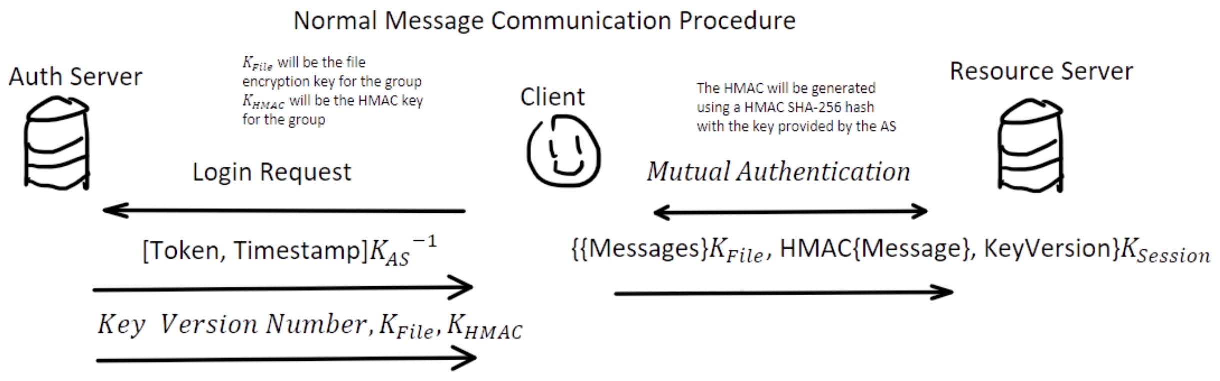
The threat outlined in this section–Private Resource Leakage–primarily consists of resource servers leaking private resources to unauthorized principals. To prevent this, we will create a system in which resource servers–and thus, anybody they leak the resources to–cannot access the resources that pass through or are stored on the RS. Only individuals authorized by the AS may access them (even in the event they are leaked!).

To create such a system, we will change our model such that, for each group, all messages that pass through the RS are first encrypted by a private symmetric key (a file key) only available to clients in that group. This symmetric key will be distributed by the AS when a user receives their token; for users that are part of multiple groups, they will receive multiple symmetric keys (specifically, one key per group / file. That way, resources that are leaked cannot be decrypted by clients from different groups). In short, the messages will be end-to-end encrypted for the clients.

The symmetric keys used for this process would be of almost the same type as the symmetric key used between the client and RS. Specifically, two 128-bit keys, the first one for AES encryption using CFB mode for the files and another for a HMAC, each key set will be assigned a version number. As we believe this bit length better suits messages sent between clients. Moreover, CFB mode works well with this model in case the clients are only sending shorter messages, in which case no padding is necessary for encryption. Of note, there will be one set of these keys per group, which is equivalent to one group existing per message history file stored on the RS. Even if there were two groups that contained the exact same members, they would each use a different symmetric key–because the AS doesn't store a list of each group's members, rather it stores all the groups a given user is in (in their permissions). For more clarity, the AS stores a unique group ID for each group that exists: each group ID will have its own associated symmetric key. Each chat message will also have a version number (in plaintext) attached to it, and the number can be used to determine which key set is supposed to be used to decrypt the message.

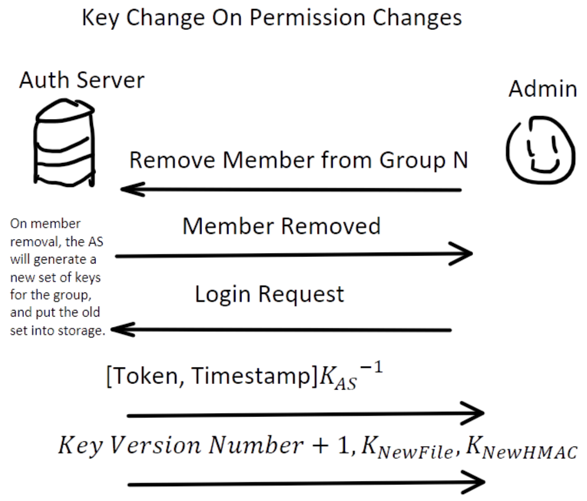
In this system, clients will send encrypted resources to the RS, which will then distribute these (encrypted) resources to every other client in the group. Then, the clients would handle the decryption of the resource. In our application specifically, this would consist of messages within groups. Furthermore, the RS will store these resources in the encrypted format; in fact, the RS won't have access to the key in question. Of note, the messages will still be encrypted with the symmetric key generated during the client-RS handshake. The RS session key is still utilized to prevent replay attacks, and to ensure the file key can be used for as long as possible. Keeping the session key will allow for the extra layer of protection by allowing end to end encryptions. As AES encryption is very fast, decrypting a second time doesn't pose much of a downgrade in performance.

I mentioned before that there would be two 128-bit keys: the first key in this system would be the key used for encryption explained above; every message sent between the client and the RS–that is meant to be private, i.e., only known to members of the group–will first be encrypted with this key. The second key in this system is used to provide each group message (only the ones not meant for the RS's prying eyes) with a unique HMAC. The HMAC will be using SHA-256 as the hashing algorithm as it is fairly common . The purpose of this HMAC is to verify if the current file key is correct, which will be explained in more detail below.

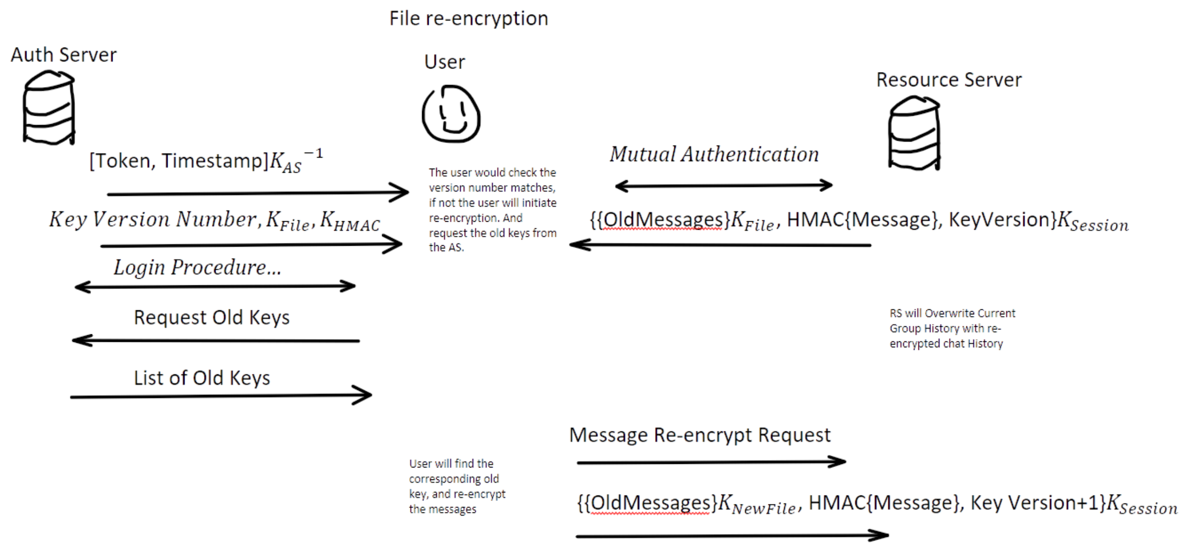

Normal Message Communication Procedure
Auth Server
K_{File} will be the file encryption key for the group K_{HMAC} will be the HMAC key for the group
Client
The HMAC will be generated using a HMAC SHA-256 hash with the key provided by the AS
Resource Server
Login Request
Mutual Authentication
$[\text{Token, Timestamp}]K_{AS}^{-1}$
$\{\{\text{Messages}\}K_{File}, \text{HMAC}\{\text{Message}\}, \text{KeyVersion}\}K_{Session}$
$Key\ \ Version\ Number, K_{File}, K_{HMAC}$

Upon any permission change in which a user **loses permissions** to a group, the AS will generate a new set of keys, and a new version number (version number+1) for that group (it is not necessary to update the key when a user is added, because we can just give them the old key). These new keys will be given to the user to encrypt all future messages.

For re-encryption (diagram on the next page; also note that this will be an ideal situation with no file tampering on the RS' side, tamper handling will be explained later), whenever an user logs in to the RS with a new file key, the RS will send them the message history of the group. Then they will check whether the message's version (included in the history) matches the current key version. If it does, the client will then try to decrypt the message. If not, the client implementation would notify the user that the chat history on the RS has not been re-encrypted yet, and ask to start a re-encryption process. At this point, the user would open a temporary session back to the AS to perform a request for the old keys. To which the AS will send the list of keys previously used to the user (the


Key Change On Permission Changes
Auth Server
Admin
Remove Member from Group N
On member removal, the AS will generate a new set of keys for the group, and put the old set into storage.
Member Removed
Login Request
$[\text{Token, Timestamp}]K_{AS}^{-1}$
$Key\ Version\ Number + 1, K_{NewFile}, K_{NewHMAC}$

whole process will still be encrypted, just not shown on the diagram for visual simplicity). The list of keys would remain at maximum 50 sets of keys (reasoning explained in the next paragraph). After receiving the keys, the user will be able to find the corresponding old file key that can decrypt the message history, then send a request to the RS to re-encrypt the chat history. After the request is confirmed, all other users will be logged out by the RS to be notified that a file key change is happening, and they would need to retrieve the new key from the AS. Upon receiving all the file keys, the client will find the version of the file key for the existing messages. The client will verify the key is correct by verifying the HMAC of the message using the specific key version. If successful the client will decrypt it with the old file key and re-encrypt with the new set. During this process, each message will be re-encrypted and have a new HMAC generated for it with the new HMAC key. Also the version for the message will be updated to reflect the re-encryption (notably, this new version is provided by the AS). Following which the client sends the re-encrypted chat history to the server for storage. The key retrieval could be done before connecting to the RS in the case where the admin that performed the permission change is re-encrypting the chat history. (Options regarding tamper handling will be on the next page)


File re-encryption
Auth Server
User
Resource Server
[Token, Timestamp]K_{AS}^{-1}
$Key\ Version\ Number, K_{File}, K_{HMAC}$
The user would check the version number matches, if not the user will initiate re-encryption. And request the old keys from the AS.
$Mutual\ Authentication$
{{OldMessages}K_{File}, HMAC{Message}, KeyVersion}$K_{Session}$
$Login\ Procedure...$
Request Old Keys
List of Old Keys
RS will Overwrite Current Group History with re-encrypted chat History
Message Re-encrypt Request
User will find the corresponding old key, and re-encrypt the messages
{{OldMessages}$K_{NewFile}$, HMAC{Message}, Key Version+1}$K_{Session}$

The reasoning behind having a list of used keys is that the file re-encryption will not happen until a user logs into the RS; as users can use any RS for their group messaging, it is very likely that some RSs will not have their files re-encrypted before the access permission for the group is updated a second time. So a list of keys is kept to ensure some leniency in re-encrypting with the server. However, the authentication server cannot hold all the keys forever, so there needs to be some point where the keys are not worth it to be held on anymore (and it's highly undesirable for the AS to send a list of potentially thousands of keys to the user). We drew the line at 50 keys as it would be very unlikely a RS' chat history will still be of use to the group members if 50 permission updates have happened without some member accessing the RS. In that case the messages on that RS will be unretrievable. For the same reason all users can initiate the re-encryption process as it is unreasonable to assume the admin will know all the RS which the group has a message history on. So the responsibility for re-encrypting the chat history in a timely manner will be left on the users.

The HMAC exists so that the client can be sure on whether or not the key is correct, as there could be padding related issues that cause failed decryption. During the re-encryption process, if the HMAC verification fails, the client will then try all other keys in the list provided by the AS. If some other key successfully verified the HMAC, then the user will try to decrypt with that key set. If none works, the user will be prompted [Possible File Corruption] and then can either check the file is actually corrupt by trying to decrypt a message with the list of AES keys, or abort connection.

The point of the system is that the RS could possibly tamper with the existing resources. As the version number is stored in plaintext, it would be easiest for it to tamper, but it could in theory also intentionally corrupt the messages by tampering with either the HMAC or the message itself. If the version number is tampered and nothing else, client's time would be wasted trying other keys, but no real damage is done. However, if either the HMAC or encrypted message is tampered, the process will become fairly complicated. We tried to design the system so that the user would still have a chance to recover the messages if only the HMAC is tampered with but not the encrypted messages. Nevertheless, in the case where either the version number, HMAC, or the message itself is tampered with, the client would be notified [Possible Resource Tampering], notifying the user that the RS would not be safe to continually use.

The protocol above will defend against private resource leakage by taking RS access of the resources out of the equation, leaving the RSs solely as a storage and distribution source. However, we must acknowledge the assumption that the RS will perform the re-encryption request as intended: in the case of a key change, if the RS does not truthfully replace the chat history with the re-encrypted file, it is possible for the RS to leak old messages to already removed members. But we deem this not an issue as the removed members already had access to the old messages (since they would have the key if not removed), and that the RS cannot leak any new resources sent after re-encryption.

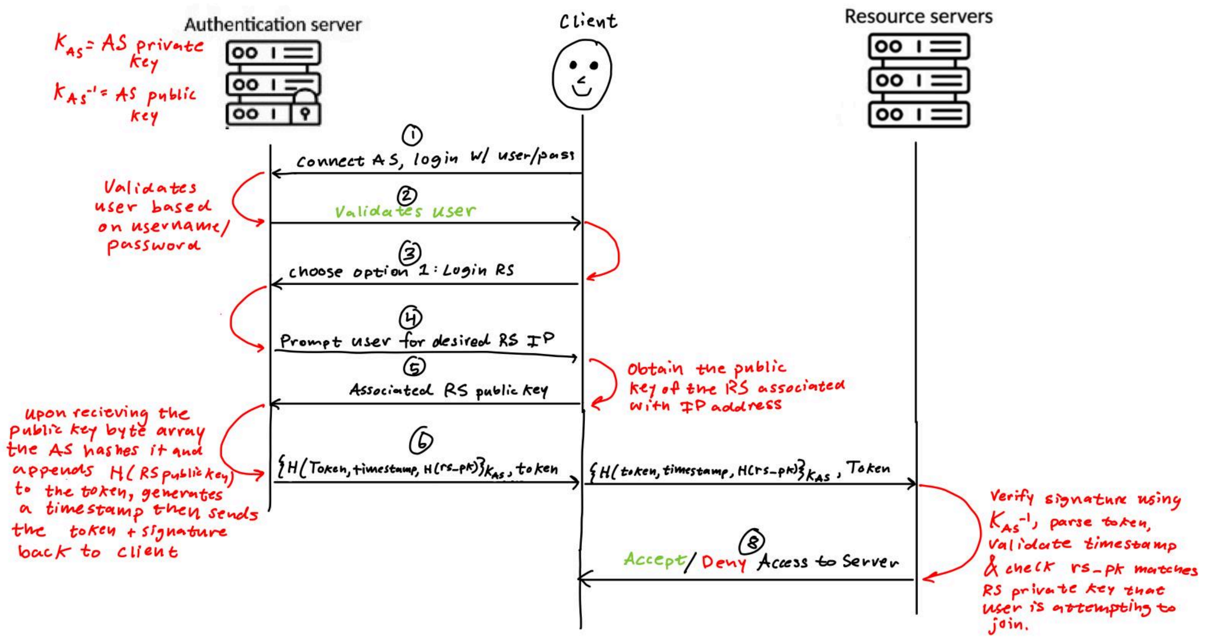
Threat 7: Token Theft

Token theft occurs when a rogue Resource Server (RS) steals a valid user's token and passes it to unauthorized users, allowing them to access the legitimate user's resources such as group chats and messages. In the current implementation, tokens do not contain any information about the specific RS they are intended for, meaning a rogue RS can distribute a valid token to unauthorized users or other servers within its validity period. This lack of binding between tokens and resource servers creates a significant security vulnerability that must be addressed to ensure the confidentiality and integrity of user resources.

To mitigate this threat, we propose augmenting the token with an additional attribute that binds it to the specific RS where it will be used. Before issuing a token, the Authentication Server (AS) will prompt the client to select the intended RS by providing its IP address. The IP address will then be used to obtain the associated RS public key which will be sent back to the AS and hashed using SHA-256 (just so we aren't appending a massive byte array to the token) and it will be appended to the token as a way of identifying a specific RS. The token will also include the username, user permissions, and expiration time. Once the token is prepared, the AS will sign it using RSA with a 4096-bit key, SHA-256 hashing algorithm, and PSS padding and the AS private key. The signed token is then sent to the client, who presents it to the RS after login.

Upon receiving the token, the RS will verify its authenticity by checking the RSA signature against the AS's public key. Additionally, the RS will parse the rs_pk (resource server public key) attribute hash its own public key using SHA-256 and check that against the rs_pk attribute and ensure it matches the newly generated hash. If the key does not match or the RSA signature is invalid, the RS will reject the token and deny access. This ensures that even if a rogue RS intercepts or steals a valid token, it cannot distribute the token to other unauthorized users to use it on different servers and gain access to their resources.

The proposed mechanism effectively prevents token theft by binding tokens to specific RS public key. Even if a rogue RS manages to steal a valid token, it cannot distribute it to unauthorized users or other servers because the rs_pk attribute restricts its usability to the specified RS. Using RSA with a 4096-bit key, SHA-256, and PSS padding ensures the authenticity and integrity of the token, making it computationally infeasible for attackers to forge or tamper with the token. The AS, being fully trusted, guarantees the accurate issuance of tokens with the correct rs_ip attribute, while the RS validates the token's signature and server binding before granting access. By introducing these checks, the proposed mechanism nullifies the threat of token theft and ensures the secure operation of the system. An example of this exchange will look as follows (all communication between authentication server ←→ client and client ←→ resource server are encrypted using AES as described in the previous phase):


Authentication server
Client
Resource servers
K_{AS} = AS private key
K_{AS}^{-1} = AS public key
① Connect AS, login w/ user/pass
Validates user based on username/password
② Validates user
③ choose option 1: Login RS
④ Prompt user for desired RS IP
⑤ Associated RS public key
Obtain the public key of the RS associated with IP address
upon recieving the public key byte array the AS hashes it and appends H(RS public key) to the token, generates a timestamp then sends the token + signature back to client
⑥ $\{H(\text{Token}, \text{timestamp}, H(\text{rs\_pk})\}_{K_{AS}}$, token
$\{H(\text{token}, \text{timestamp}, H(\text{rs\_pk})\}_{K_{AS}}$, Token
Verify signature using K_{AS}^{-1}, parse token, Validate timestamp & check rs_pk matches RS private key that user is attempting to join.
⑧ Accept/Deny Access to Server

Conclusion

Overall, we have developed mechanisms that we believe will address these 3 threats while still protecting against the threats from phase 3. These new mechanisms require very minimal change to our existing mechanisms, ensuring all 7 threats are accounted for. For example, messages will still be encrypted as they were before, just with an additional encryption added to prevent the RS from having access to unencrypted messages. The mutual authentication between the client and RS will follow the same steps as previously, just with an additional check by the RS that the token's RS hashed public key attribute is correct. We did modify some of these mechanisms after seeing feedback for another group who had similar ideas. For threat 6, we originally planned on giving admins the responsibility of logging on to resource servers to notify them of a key change; we changed this to automatically notifying users when messages need to be re-encrypted so that admins would not have to know about and be responsible for every RS.

Additionally, for threat 7 we originally planned on including the RS’ IP address in the token but changed this to the server’s hashed public key.