In our current system, we have no good protections against any of the threats identified in phase three. Thus, we have come up with methods to properly address all four of these threats. The first threat, unauthorized token issuance, will be addressed by checking user passwords such that a user will need to provide the correct password for a username in order to obtain the token for that username. Additionally, we plan on using Diffie-Hellman protocol and AES encryption to encrypt the password and token when they are sent between the client and authentication server (AS). To address the second threat, modification and forgery of tokens, we plan on attaching an RSA signature to a token whenever one is sent from the AS to the client or from the client to the resource server (RS). For the third threat, unauthorized resource servers, we plan on implementing a mutual authentication protocol between the client and RS based on the SSH protocol; the RS will need to know the correct private key to be authenticated in the protocol. Finally, there is the final threat (which is addressed to a certain degree in our solutions to the previous threats), information leakage by passive monitoring. We plan on ensuring that all communication between the client and the AS or RS is encrypted in some way. Additionally, important messages such as tokens will include a signature to prevent impersonation. We will now address each of these threats and our solutions for them in greater detail. But before that, our threat handling relies on a number of key assumptions listed below:

Assumptions :

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.
    a. To ensure the trustworthiness of this platform, we (the developers) would be responsible for running it and ensuring the data uploaded is trustworthy. Similar to how we are designing this current system, we would employ the necessary cryptographic / cybersecurity protocols to ensure users may only log in to their account, and modify information for servers they own.
        i. To ensure that impersonations do not happen, we could similarly use a password authentication protocol, possibly with the additional step of verifying the server's private key (that way, an adversary who steals ONLY the private key would not be able to log in without the password, neither would one who cracks ONLY the password).
        ii. When a user wants to upload a public key for a server they own, we could verify that both the server uses that public key (by manually confirmation, i.e., trying to access the specified server with that public key) and the user uploading knows the private key (to ensure that they are able to respond to client requests made with the uploaded public key)

b. In an actual implementation of this system, we would have the client perform https requests to this trusted website to obtain the necessary public keys. However, as this website won't actually exist, we will instead store public key information locally on the client-side (which the client can manually enter into a file).
c. Justification: The nature of our application (i.e., that users need to manually input server IPs, port #s, and public keys) necessitates some methodology for users to learn the IPs, port numbers, and public keys of the servers. Leaving this responsibility to the users or server owners would result in a major inconvenience for both server owners (who would have to find some way to not only publicize their information, but also manage it and curb disinformation) and users (who would need to search far and wide to even find servers to use). This would heavily discourage users and server owners from using our application; thus, our solution is an associated (trusted) platform in which server owners can upload their IP, port number, and public key, and users can easily view this information, making the use of our application convenient for both users and server owners.
    i. This is a reasonable assumption to make for our system because there will be many users of our system that wish to join public groups and communicate with those they may not know. For private groups between friends, they can easily share the relevant information among each other; however, for groups dedicated to specific niches or hobbies, they will want some way of learning what groups exist and how to join them
    ii. Many websites like this already exist for various different applications where users can host their own servers. Discord, which is somewhat similar to our application, has a website listing public servers and information about them (i.e., if they're verified, links to join the servers, etc.). Even games like Minecraft have websites that organize and list server IPs for users to connect to.

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.
    a. Secure storage and protection of private keys implies that the servers, to the best of their abilities, will ensure that their private key is not shared with or accessible to others, and if they discover that their key is leaked, they will take the necessary action to prevent any impersonation (by generating a new key pair). This assumption does not require any specific implementation on our part; rather, it is an implicit assumption that the servers will act rationally with the goal of maintaining the secrecy of their private key. This could entail using secure environment variables, an encrypted file system, or any other methodology of protecting their key. Furthermore, this assumption also implies that the server owner(s) wouldn't be negligent with their protection, such as leaving the private key lying around unsecure on a piece of paper.
    b. Justification: This assumption is made to ensure that no adversary could impersonate a server by using their private key in any of our protocols. It is standard practice to assume that owners of private keys will protect their private keys; otherwise, there's no point to encrypting anything in the first place. But

mistakes happen, which is why we allow for servers to alter their private key as necessary.

3. The authentication server is completely trusted.
    a. Justification: Provided in the phase 3 document.
4. The attacker has full read access over the communications channel between servers and clients, but cannot directly interfere with the transmission process itself. Essentially, the attacker has no way to perform most active attacks, i.e. session hijacking. The only exception is that the attacker can attempt to impersonate users and resource servers.
    a. Justification: Provided in the phase 3 document.
5. There is a universally accepted and known large prime number q and generator g used for Diffie-Hellman key generation for both AS and RS communication.
    a. Justification: As designers of the implementation of our design, we will be able to hard-code these values into the implementation of our servers and clients.

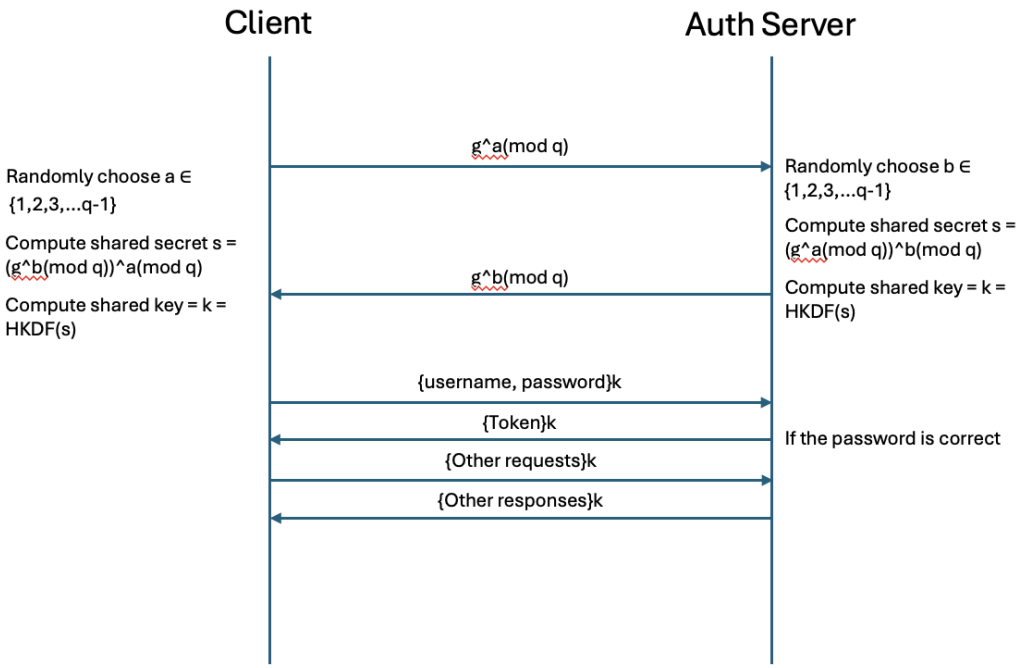
For threats that require public key cryptography, we will employ RSA SHA-256 with a 4096-bit key size and PSS padding. SHA-256 will be used as our hash algorithm due to its robustness and suitability for cryptographic protocols. A 4096-bit key size is used to prevent a brute force attack and ensure enhanced 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.

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

Threat 1: Unauthorized Token Issuance

This threat is characterized by giving a token to the wrong actor, either by deceit on the actor's part or any other mechanism that allows users to obtain tokens for an account that they do not authenticate themselves as. In our system submitted for phase 1, due to the assumption that the user was completely trustworthy, the user authenticated themselves with the AS via their username, which is public information. The client does request a password that is sent in the AuthRequest to the AS, but this was not used for authentication. If an attacker wanted to attempt an unauthorized token issuance attack on our old system for "User1" who has the password "Pass1," (which was unknown to the attacker) the attacker would merely sign in on the AS with "User1" in the username field and any string for the password. The AS would accept this, and return the attacker the token, which the attacker could then use to access the user's messages. This violates the user's assumption that their conversation content would be private and that they would not be able to be impersonated.



The proposed mechanism we would use to approach the unauthorized token issuance attack is by checking the passwords on the AS side. We would want to protect sensitive information (the password and the returning token) from snoopers by encrypting these exchanges using Diffie-Hillman. We will use a 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 key (though the expansion won't be necessary because the shared secret would also be 256-bit). We will then have the AS generate a random initialization vector that will be transmitted unencrypted to the client so that they can exchange information. For AES, we plan on using a 256-bit key due to the added security and the fact that the amount of data being encrypted is limited to requests and tokens, which are small enough that the performance impact of using a larger key should be mitigated. We are also planning on using CBC mode because the randomization will prevent identical plaintext requests from having identical ciphertexts. This would prevent a snooper from being clued into the fact that the user is acting differently, such as changing their password or signing into a different account. This could prevent some social engineering attacks a potential snooper could use. We also propose to use a PKCS7 padding because it is a pretty common standard and padding is necessary for AES.

In comparison to our Phase 2 solution, we want to change our AS-client interactions so that not only one request is being handled per connection. Rather than closing a connection after a request is handled, our Phase 3 system will maintain a connection with the AS for multiple interactions. This will allow us to use the same AES shared key for all interactions.

On the AS, we will search the hashmap containing the username and password pairs to see if the pair supplied by the user is a match. We keep these passwords saved on the AS because of the assumption that the AS should be completely trusted.

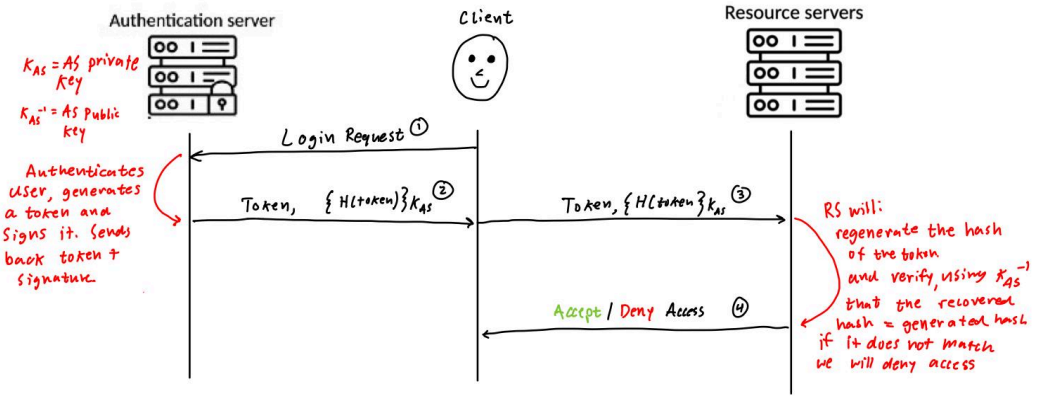
The protocols and mechanisms described above should adequately prevent an attacker from obtaining the user's token because the AS checks if a client has the correct password before issuing the token, which prevents any attacker from impersonating the user. Additionally, the encryption of both the password to the AS and the signature on token to the client should prevent any snoopers from listening in and stealing the information from either to use to sign in to the resource server.

Threat 2: Token Modification / Forgery

Token modification/forgery involves unauthorized changes to a user token which contains their username and group they belong to with their permissions for said group, it is sent by the AS upon authentication and used on the RS side to grant permissions/privileges to a user based on the token sent from the RS. This is a threat because an adversary could theoretically obtain this token, make changes, and be able to access group chats that they 1. Previously did not have access to or 2. Did not have administrator privileges for (i.e., they could not add or remove users and could not delete the group chat). In our old system, tokens were sent to the client from the AS, upon authentication, as a token object that is serialized into a JSON string, the client then sends the token to the RS where it is deserialized and parsed to grant permissions. An attacker in our previous system could simply obtain the serialized token over an eavesdropped communication channel, deserialize it, and then append (group number, permission) codes to their token before serializing it again and sending it to the RS. The RS has no way of knowing this token was modified and will thus parse it as normal and grant unauthorized access based on the modified token.

To prevent this kind of attack from occurring we will use RSA to sign tokens being sent from the AS. We will use RSA to create a digital signature on a token generated by the AS, RSA is the best method for preventing this attack due to its robustness and public key infrastructure only requiring distribution of a public key and not requiring any shared secrets relying on assumption 1. We will also implement a timestamp for the tokens to prevent replay attacks from occurring. This will be generated when the token is created before signing.

A diagram of this protocol will look like this:



The user will send a login request to the AS, if the credentials are valid, a token is generated for that user and then signed using RSA (4096-bit key, PSS padding, and SHA-256 hashing algorithm see paragraph below the assumptions for why) and the AS private key (K_{AS})

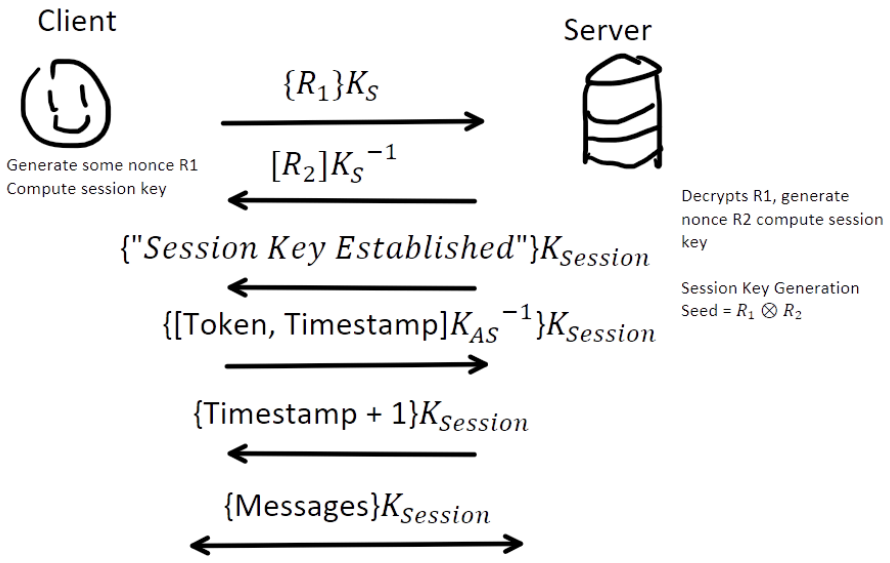
for signing the token. The token is then sent back to the client along with the signature. The client will then be redirected to the RS, where the signature of the token will be verified using the AS public key (K_{AS}^{-1}). The RS will regenerate a hash using SHA-256 of the token and verify the signature and compare the results. If there is any discrepancy in the outputs of the recovered hash we know there was attempted modification of the token and we will thus deny access and invalidate the token.

This mechanism is sufficient in preventing token modification because of the signature that is created by the AS using a private key. The RS will use the AS' public key to verify. If the verification fails it will not accept the token or grant any permissions based on said token. The RS will also check the current time against the timestamp and current time, rejecting tokens that fall outside the valid time range. Any attempts to modify the token by changing any permissions or the timestamp will require the adversary to obtain the AS private key (and resign the token) which is computationally infeasible to recover from the any eavesdropped information obtained based on what we know about RSA using a 4096-bit key, SHA-256 hashing algorithm, and PSS padding scheme. This protocol will ensure authenticity and integrity that it was sent from the AS and not modified.

Threat 3: Unauthorized Resource Server

An unauthorized resource server attack is a very powerful attack; impersonating a server and attempting to steal a user's secrets could completely compromise said user's account. For example, in our system, an adversary that impersonates a resource server could easily steal and parse a user's unique login token and use it to sign in to their account. To prevent an unauthorized resource server attack from happening, we have adopted a protocol for connecting to resource servers based on the SSH protocol, a standard method for initializing secure remote connections. This protocol relies on assumptions (1) and (2), detailed above.

The protocol we have designed for connecting to a resource server is as depicted in the diagram below (for more information on the protocol—like key sizes or type, encryption algorithms, and why we made the choices we did, see T4):



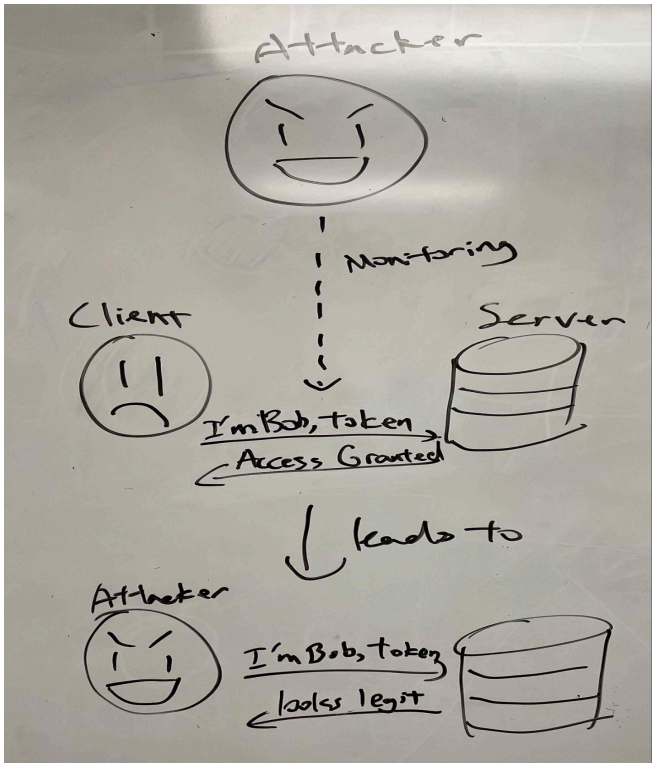
This protocol protects against an unauthorized resource server attack thanks to the fact that it verifies the server's unique private key, which, as assumed above, is known only by the server themselves. To decrypt $\{R1\}K_S$ and sign R_2, an attacker would need the server's private key, which is assumed to be known only by the server. This protocol simultaneously allows

the server to verify the user thanks to the user's specific token, which is unique to each user, only available to the user themselves, and signed by the authentication server with a timestamp (so if a user gains access to the token itself, the RS won't accept it without the timestamp and token signed by the authentication server itself). Furthermore, since future communications in this session will be encrypted with the session key, there is no worry of a man in the middle attack–snooping on confidential information or hijacking the session. Of note, if anything goes wrong on either side (e.g., there's a key mismatch, the verification fails, the token decrypts to junk, R2 isn't signed, or the token isn't signed), then the side that notices the failure will immediately terminate the connection. However, if the timestamp is too old, then the server will send back a message informing the user that the timestamp is invalid (instead of sending back the timestamp+1).

This protocol also protects against replay attacks by generating the session key with nonces from both the client and the server. Since R1 are sent encrypted to the server, even if an adversary were to replay a previously used R1, since these are both encrypted with the server's public key, there would be no way for this adversary to know their actual values. Thus, they would be unable to compute the session key—rendering them unable to parse any messages. However, the protocol would end long before this point, because replaying a previous token would not only fail due to the session key being different (thanks to the key being generated with the server's randomly generated R_2), but would also likely fail due to an expired timestamp (unless the attack is replayed almost immediately, in which case the former would cause the protocol to fail).

Threat 4: Information Leakage

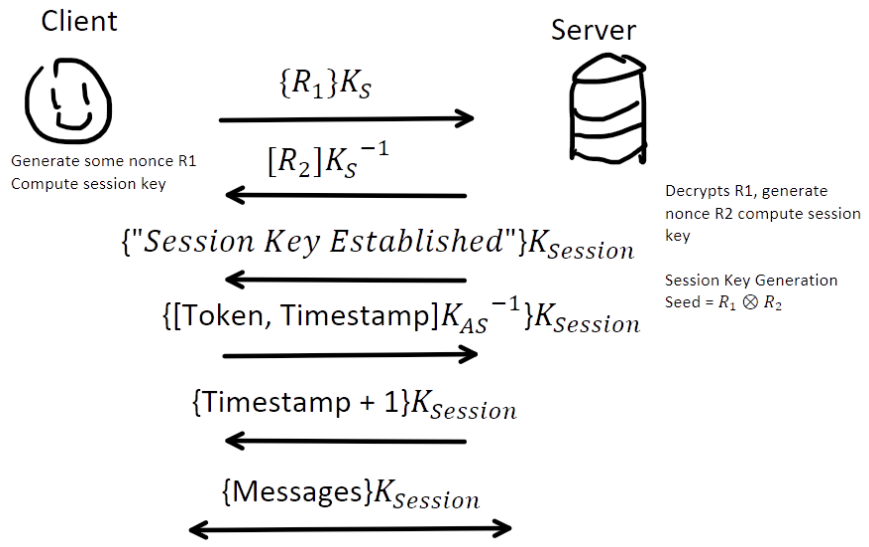
The threat here is described as an attacker having read access over the communications channels (See assumption 3 for specific details). This would allow the attacker to monitor the communications channel and retrieve information transmitted through these channels. Then the attacker can use the information obtained through the monitoring process to impersonate users and servers. For example, assuming a system without any authentication or encryption, if a token is sent to the server, then the attacker could just record the token then use it to impersonate the user who sent it.



In this section, the focus will be on the specific mechanism implementation for the resource servers (RS for short). This same problem could occur with the authentication servers (AS for short) communications, but the mechanism described in Threat 1 will defend against the passive eavesdropper. So for the AS specific implementation, please revisit the section on threat I, as the section does describe how the mechanism defends against both impersonation and eavesdropper. So it would not be repeated here. As the AS will prioritize

confidentiality, the specific implementation choices such as AES mode of operation, key sizes, and padding might be different than what is used in the RS. The RS will take some tradeoff in security for better performance, nonetheless the choices here will still be secure against the attacks, just less secure compared to the AS due to smaller key sizes, etc.

When trying to establish communication with a RS, the client will generate a 32 byte nonce R1, and encrypt it with the RS' public key K_S. The result will be sent to the RS for decryption. The RS would decrypt the content, generate a 32 byte nonce R2, and sign the nonce with its private key. The choice for a 32 byte nonce and secret is simply a good length for a random, one-time used number (256 bits) as it is safe to assume the attacker would not be able to brute force the size in a reasonable time. At this stage, the RS should also generate the session key. By XORing R1, and R2 together, the results would be used as the seed to a secure random generator to generate a 128-bit AES key. To be used with an AES cipher with CFB mode. Choice for AES is for the security of the encryption method and CFB for its performance in encrypting user input. The key has a 128-bit IV and no padding, as the IV size matches the block size of the cipher. As padding is used to fill in the difference between the ciphertext and the plaintext, CFB mode will not need padding. The reason being that the ciphertext in CFB mode is the result of XORing the plaintext with some output of the encryption function, so the size of the plaintext and the resulting ciphertext will be the same (See citation at the end for source). The key will be discarded by both the client and server after the session ends. The 128 bit key size choice is simply because 128 bit is considered secure in production, and as it is for one time use, efficiency would be a priority in key generation.



After R2 and the session key is generated, the server would then send a message to the client alongside the R2 nonce encrypted with the session key to signify the end of the key exchange. At this time the client can also compute the key using R1 and R2. The server will store the key on a hashtable in the server backend alongside all other client's session keys, and when the client disconnects, the server will remove it from the list. The hashtable exists because each server thread is also responsible for broadcasting the incoming messages to other users. The messages broadcasted to different users needs to be encrypted with different session keys, and the server needs the table to properly encrypt the messages as the session key will not be the same for different users. The key then can be used to encrypt and decrypt messages sent between the client and the server. After the client computes the session key, the token is sent, encrypted with the session key. The token and a timestamp is signed by the authentication server. The RS

would reject the request if the timestamp is too old. The token time availability is set to 5 minutes, for we expect the users to reauthenticate every time they wish to connect to RS, and the user would attempt connecting to the RS immediately after authenticated. After the RS decrypts all information, it sends an updated timestamp to the client, encrypted with the session key, signifying that the authentication is successful. The new timestamp should be within 6 minutes of the token issuance, giving some leeway for users using the token at the last minute during high server load. The timestamp sent back by the server is for safety against a reflection attack. Then afterwards all message exchanges between the client and server would have all information encrypted by the session key. During this entire process, if the RS cannot verify any part of the exchange (i.e. verifying AS signature failed, or failed to decrypt message via the session key), the RS will consider the handshake failed and abort the connection.

The mechanisms would block the passive attacker from retrieving any useful information for impersonation purposes. The mutual authentication would block the attacker from impersonating as a RS, as it cannot decrypt the nonce correctly, so the session key generated by the RS and client will not match. The AES session key will prevent the attacker from learning anything from the messages, and no one has broken 128-bit AES encryption with brute force yet, so we deem it secure enough for the usage. Replay attacks will be eliminated as session keys are generated with different nonce's everytime, so that the attacker connecting to the server cannot resend the same nonce as the client to retrieve the same session key. Without the same session key, the attacker will not be able to impersonate the user. All information after the key exchange is all encrypted by the session key, so the attacker would not be able to read anything sent by the server even if they log in, nor send anything back as they cannot encrypt their messages. If the session key is cracked somehow, the attacker would obtain information on the user's token, but it would not help the attacker impersonate the user, as it is extremely unlikely the attacker will crack a 128-bit AES key within the allotted valid time for the timestamp. To tamper with the timestamp would require obtaining the AS' private key in some way, either by compromising AS or brute forcing the key, neither is within the scope of this phase.

Conclusion

Overall, we have developed a system that we believe adequately addresses these 4 threats. However, there were some elements that we originally planned to be part of the implementation that ended up being scrapped. For example, we originally planned on attaching an HMAC to tokens as part of the mechanism for threat 2, but scrapped this idea as it would require all resource servers to have the same private key. Additionally, the mutual authentication protocol between the client and RS was skimmed down from its original design for efficiency reasons. Furthermore, while our design process aimed to create an individual solution for each threat, in the end there is a lot of overlap between mechanisms, particularly between the mechanisms for threats 1 and 3 and threat 4. This is because, to properly address threats 1 and 3, the first threat of a passive attacker needed to be considered; this meant that our mechanism for threat 4 essentially ended up being part of the mechanisms for part 1 and 3.

Sources:

NIST SP 800-38A, Recommendation for Block Cipher Modes of Operation, **Appendix A**
https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38a.pdf

NIST SP 800-78-5, Cryptographic Algorithms and Key Sizes for Personal Identity Verification, **Page 7**
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-78-5.pdf