Phase 3 Mechanism Deliverable

The following content describes in detail the mechanisms for which unauthorized token issuance, token modification/forgery, unauthorized resource servers, and information leakage via passive monitoring threats are protected against in our implementation. Each threat is outlined with threat descriptions, the prescribed mechanism to protect against said threat, and a thorough defense for why this mechanism was chosen.

Assumptions

Many of our protocols will expect servers to know the public keys of other servers, namely the single Authentication server. To meet this requirement, we assume the existence of a secure key distribution server that any server will be able to access and see the public keys of each server. This key distribution server will be maintained by the root user, so we reason that it is trustworthy and always contains correct information.

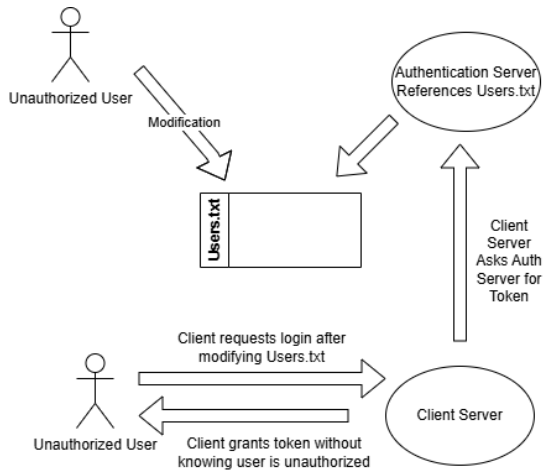
Threat 1: Unauthorized Token Issuance

Before describing what an unauthorized token issuance is, we should first define what entails an unauthorized versus an authorized user. In our implementation, an authorized user is one that is able to successfully login to the client server. If the user is able to login successfully, this means that their username and password are registered in the authentication server's master list of usernames and passwords. However, a user can be considered “authorized” if they are able to login and a username and password exist, but they are unable to execute any commands if they are not a part of a group (excluding root user). Therefore, *a fully authorized user is a user that is able to successfully login with a username and password that the authentication server has stored, and is a part of a group under the condition that they do not have access to the root user’s credentials*.

Taking this a step further, an unauthorized user that obtains a token can take multiple routes to do so:

1. Stealing/modifying the file that stores the usernames, passwords, and groups of all authorized users
2. Listening over a network and stealing incoming/outgoing tokens

Figure 1: Unauthorized user modifies master list of users



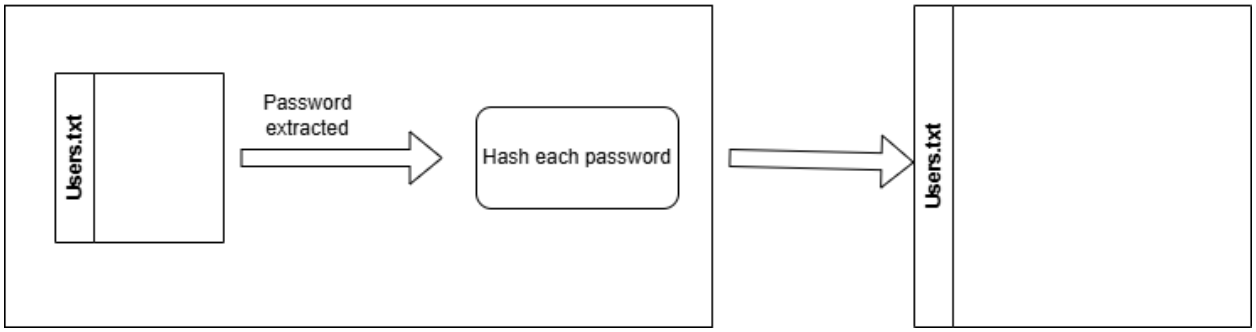
Mechanism 1: Generate Symmetric Session Keys via Handshake

The threat of tokens being stolen over a network is acknowledged in Threat 4: Information Leakage via Passive Monitoring, where the Authentication Server and the Client Server implement a handshake protocol with symmetric key encryption to generate a session key. In our previous implementation, our Client prompted a simple username and password system in which the user was then granted a token for the duration of their login. This token would be the same even if the user logged in again.

In our new implementation, we would like to implement session keys, in which the authentication server, and in turn the key, is only active for the duration that the user is active. For additional detail, please reference T4.

Mechanism 2: Hashing user information

In this mechanism, our auth server holds a file that contains the information of the usernames, passwords, and groups that each user belongs to. When the user is authenticating, the authentication server references this file in order to see which users are valid. Upon login attempt, a password will be hashed with bcrypt and a user-specific salt. The authentication server will compare the computed hash to the hash that it has stored in an arraylist (generated from a file of user information) when authenticating a user. The Users.txt file will store a username, a salt, a hashed password, and a group. The salt is stored in the Users.txt file because an adversary will not have any useful information from it unless they know what the password is. We also chose to hash passwords with a salt because it ensures each user's hash remains unique even if two users have the same password. The salts will be unique to the user.



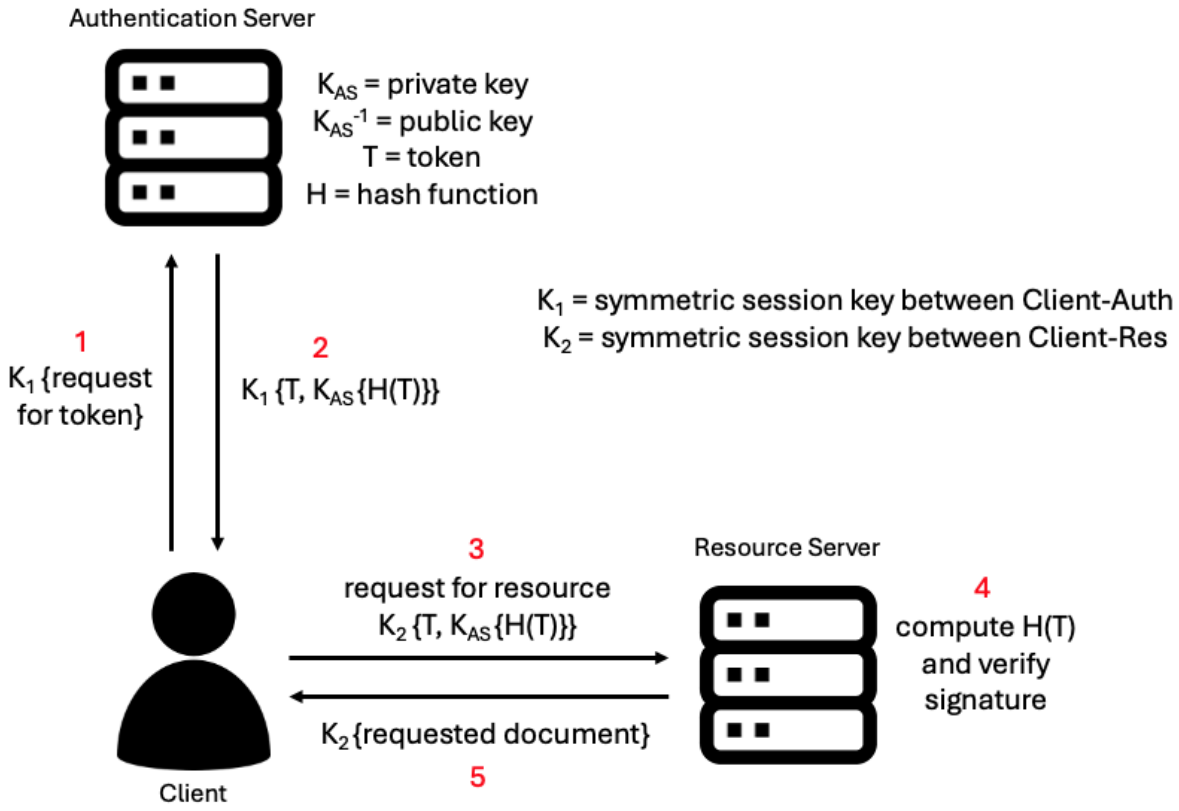
Conclusion: This mechanism will work heavily with Threat 4 to enforce that an adversary is not able to obtain a token without the permission of the authentication server. To ensure additional security, we will hash the passwords and the group in the Users.txt file so that it remains secure should the file somehow be intercepted.

Threat 2: Token Modification/Forgery

In our current system, it is quite easy for an attacker to modify a token. Tokens are sent to the client with no cryptographic securities and contain a group name that specifies which group's files the client can access. If an attacker knew the name of the group that owns the file(s) they were attempting to access, they could alter the token's group field to the desired group and gain access to their files. It is also reasonable to assume that a token could be forged quite easily as well. In our implementation, a token contains a username and group, but a Resource Server only looks at the group in order to determine the access that will be granted. Therefore, an attacker only needs to know the name of an existing group in order to access their files because the username will not be checked.

In order to resolve this issue, we can use digital signatures to prevent the modification and forgery of tokens.

Mechanism 1: RSA Digital Signature



The use of an RSA digital signature will ensure that each token issued by the Authentication Server is unforgeable and verifiable. Using the BouncyCastle API, we will implement a standard that requires every token issued by the AS to be signed using a private key known only by the AS. Resource Servers will have access to the corresponding public key, as specified by our assumptions, which will allow them to verify that the token came directly from the AS.

Additionally, we will hash each token before signing to increase the speed that tokens can be signed (and therefore transmitted). By using a widely available hashing algorithm to do so, we can ensure that the receiver of a token, likely a RS, can hash the message they receive using the same algorithm to then verify it with the hashed version that was sent by the AS. This will speed up the signature and verification process greatly because the hashed version of a message can be much shorter than the plaintext version, meaning that the RSA decryption/encryption will be much faster. To allow the token to be hashed, we will implement a toString() method that writes the contents of the token to a single string. Tokens only contain a username and groupname (two strings), so the function will simply combine these into one string.

Specifically, we will use the BouncyCastle API's signature methods to accomplish this mechanism. Initially, the AS will have generated a public-private key pair, which will be done using a KeyPairGenerator initialized with the RSA algorithm and a SecureRandom object to create keys of 4096-bit length. 4096 is a bit long for RSA key pairs, but it is acceptable because we will be using it to

sign the shorter hashed version of tokens, so the speed difference is negligible. For signing and verifying, we will use a Signature instance initialized with SHA256withRSA/PSS. The use of SHA256 provides the hashing algorithm discussed above and RSA with PSS provides a randomized padding scheme that will provide better security than one such as PKCS.

In the case that an attacker tries to modify a token that they receive, the signature will be modified and will invalidate the token. This will prevent any modification of existing tokens. In the case that an attacker tries to create their own token, they will be unable to produce a valid signature because they do not have access to the AS's private key used to sign tokens. Based on these two observations, we can assume that tokens are unable to be forged or modified.

Threat 3: Unauthorized Resource Servers

In our current model, users of the system can launch their own resource servers, so long as they have access to the source code. Furthermore, the client chooses the address and port of the servers they listen to. As such, it is impossible for the singular authentication server to know about all resource servers - the information it sends to the client is independent of what resource server the client is trying to connect to. This also means the user must have prior knowledge of the resource server's existence, its IP, and what port it listens on in order to establish a connection to it. The resource servers themselves have no way to prove their identity other than this "prior knowledge" system - unlike the client which sends a token with metadata attached. This means someone could set up a malicious resource server on the same location and steal token metadata, as well as any information the user attempts to upload to the server.

Assumptions:
The main assumption of this mechanism is that the desired resource server's public key is known by the client before connection is established. There are some grounds that make this assumption reasonable. Authentication servers cannot know about all resource servers, so clients must get the resource server's public key another way. Furthermore, in order to connect to a resource server, one must first know of its existence and location - the client application asks for IP and port information to connect to a server, but it does not know any details about the server it connects to. The system is designed with this in mind - it is a small file sharing platform. Therefore, it is reasonable to assume that if a client knows about a resource server's existence and location, they will have some out-of-band method of acquiring the RS public key before establishing a connection.

A secondary assumption of this mechanism is that all resource servers have access to the authentication server's public key. In order to verify clients to resource servers, authentication servers send signed tokens to clients. In order for resource servers to verify that signature, they must have access to the auth server's public key. Since there is one universal and trustworthy authentication server, it makes sense that its public key should be easily accessible.

Mechanism: Authentication + DH key exchange
The main mechanism for T3 will be similar to the second mechanism in T4, although it contains details specific to the RS. After establishing a connection with the RS, a client will encrypt the following:

- Token provided by AS
- Signature of a hash by the AS (as described in T2)
- Timestamp *t*

It will compute this using the AES-256 encryption scheme with CBC mode, using a new secret symmetric key K_{HI}. Then, the client will use a previously agreed upon generator *g* and large prime order *q*, of which the group size will be 2048 bits (group 14), to generate g^a mod q, where a is randomly chosen inclusively between 1 and q-1 and kept secret. The last thing the client will do is compute an HMAC-SHA256 over the ciphertext and DH public key using a second generated secret key K_{MA}.
Finally, it will send:

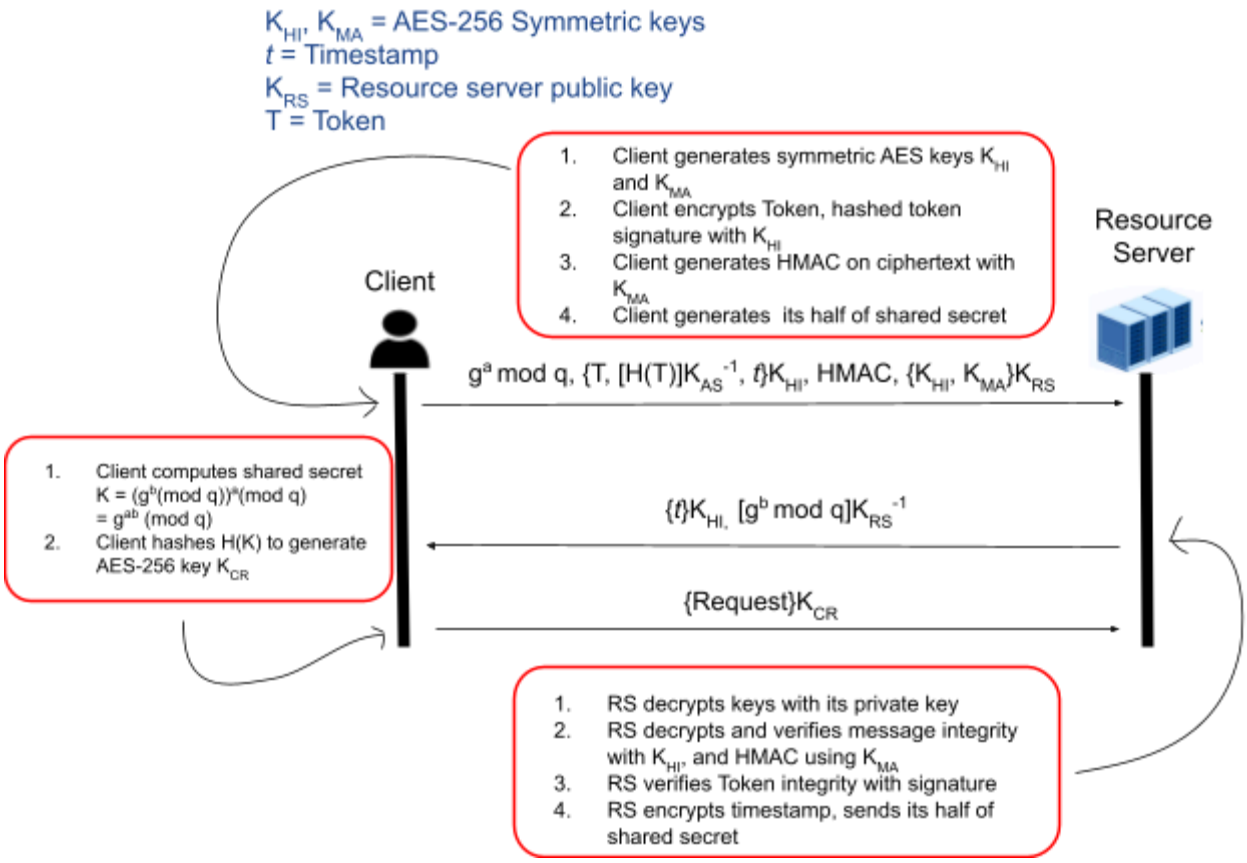
- g^a mod q
- Encrypted message
- HMAC
- $\{K_{HI}, K_{MA}\}K_{RS}$

Using a generated symmetric key for encryption and verification instead of encrypting the message using the RS public key should speed up the process of encryption and decryption, while still maintaining confidentiality and integrity.

The resource server will then decrypt the keys using its private key. Then, it can use K_{HI} and K_{MA} to verify the message was not modified in transit, and then decrypt it. From here, it can verify the token has not been modified or forged by hashing it, and checking it against the AS signature. At this stage, the client is authenticated to the RS. Then, the RS can generate its half of the shared secret with its own secret value *b* inclusively between 1 and q-1, using the same generator and large prime as the client.
Next, the RS server can send back:

- Encrypted Timestamp
- g^b mod q signed with private key

The RS is verified to the client. At this point, client and server should throw away K_{HI} and K_{MA}, and use their exchanged information to generate a shared secret. This secret can then be hashed using SHA-256 to generate the same AES-256 symmetric key for both the RS and client, which can be used as the session key.



This protocol verifies the client and auth server in two messages and exchanges a shared secret, which they can use to generate a symmetric session key to encrypt any further communications. Since the client authenticates itself with a token, it doesn't need to sign anything for proof of its identity. However, if g^a *mod q* was exchanged without a signature or integrity check, the protocol could be hijacked. Since it is included when generating the HMAC, the HMAC will verify that both the encrypted information and the public information has not been modified. Once the information sent has been integrity-checked, it can be decrypted. This decrypted information will verify the identity of the client. The signature by the AS on the hash of the token can be checked without the client sending the hash itself, since the RS has all the resources it needs to hash and verify. The resource server can then encrypt the timestamp and send it back, which verifies its identity to the client. The other half of the shared secret must still be signed in order to ensure a hijacker cannot replace the RS half of the shared secret. The timestamp guarantees freshness of the interaction and prevents replay attacks. Discarding the original secret keys makes sense because the session key will provide forward secrecy in the context where the authentication messages are compromised.

Threat 4: Information Leakage via Passive Monitoring

In our current model, all of the communications between the client and server applications are *not* hidden from outside observers. That is, if a passive adversary were to be listening in, they could obtain

information via client to server and/or server to client. In this threat model, the attacker can view data in transit but cannot alter or disrupt it. The idea is that they would monitor communications to intercept sensitive data such as file contents, usernames, passwords, and authentication tokens. This could leak user information, group information, or allow unauthorized access by capturing tokens in transit.

To defend against this threat, we will implement end-end-encryption for all communications between clients, the authentication server, and resource server.

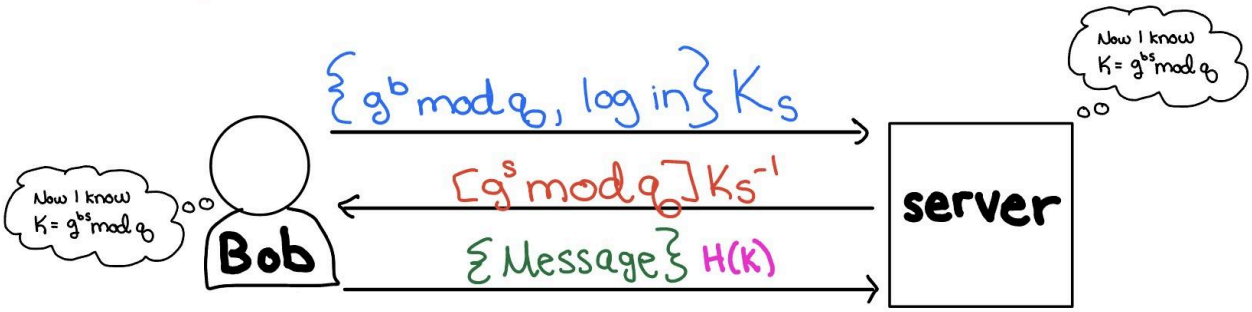
Mechanism 1: Symmetric Key Encryption (AES)
We will use AES with a 256-bit key in GCM mode to secure communications. AES is both secure and efficient when dealing with distributed networks. GCM mode provides both confidentiality and integrity, both preventing adversaries from viewing the message content, and detecting modification. Each client-server session will have a unique AES key to prevent the possibility of future key re-use.

- Key generation: Each session will begin with the server generating a unique AES key. This key will be shared using a secure key exchange method described in mechanism 2.
- Encryption: Before data transmission, each message will be encrypted using AES
- Decryption: The message will be decrypted by the recipient using the shared AES key, allowing for only the intended recipients to access the message content.

Mechanism 2: Secure Key Exchange Using Signed Diffie Hellman
To securely exchange the AES session key between the client and auth server, we will implement Diffie Hellman key exchange combined with digital signatures, and a hash function. This approach will ensure that the key exchange process itself is secure and authenticated.

- DH set up: We will use a DH group with 2048-bit prime modulus, which provides a strong foundation for secure key exchange. Both the client and auth server will generate DH key pairs. Using these pairs, they can establish a shared secret without directly sending the AES shared key.
- Signatures for authentication: To authenticate the key exchange, first the client will send the generated g^b mod q along with their username and password, and that all will be encrypted with the authentication server's public key. The server generates g^s mod q and signs with the server's private key. This ensures that the key exchange can resist man-in-the-middle attacks, as the recipient can verify the signature using the sender's public key. For example, user Bob generates g^b mod q and sends it with his login information (encrypted with the public key). The server generates g^s mod q and signs with the server's private key.
- After the signing and because they now know shared secret K, (K= g^(b*s) mod q), both the client and the server can hash it using SHA-256 to generate a 256-bit symmetric session key–that they can now use to encrypt their messages back and forth between the servers.

- Bob's DH pair and log in info encrypted with Auth Server's public key
- Server's DH pair signed with server's key
- SHA-256 Hash function



Thank you for reading our proposal!

Handshake via Client/AuthServer and Client/ResourceServer:

Each session will begin with generating a unique AES key, "K" to serve as user Bob's active session key for communication with the respective server. To securely exchange the AES session key, we will begin by both Bob and the server generating DH key pairs.

Now, first, Bob will send his portion of the DH to the server. (The server now knows Bob's DH key, so it now it knows the session key, "K".) Next, the server will send back its portion of the DH to the client. (Bob now knows the server's DH key, so Bob now knows the session key, "K".) Along with this, the server will also send Bob its portion of the DH *signed.* Why? So Bob can verify the signature of the server's public key to prove it is the real server (because in this stage, the resource server has the potential to be phony.) Additionally, the server will use the session key "K" to encrypt a confirmation message back to Bob. Why? Bob *has* to know K to be able to decrypt the message, so if someone is impersonating Bob, they won't know what that message means because they can't decrypt it. Bob will now use K to decrypt the confirmation message and see that is was a success. At this point, Bob and the server are safe to trust each other. If it's the resource server, Bob will now send his token in a message encrypted with K, and continue to send messages encrypted with K. If it's the authentication server, Bob will now send his username and password in a message encrypted with K, and continue to send messages encrypted with K.

- Bob's DH pair
- session key, K
- server's confirmation message
- Server's DH pair signed with server's key
- Server's DH pair