Group PPKJJ: Phase 4 Proposal

Introduction

Jiang

5chan is a content sharing platform that allows users to create posts and comment on other's posts if they have proper access. This phase introduces more threats and retracts a few assumptions made previously about the system, making 5chan more secure.
Building on the previous system, these techniques are added:

- HMAC (Hashed Message Authentication Codes)

The system still uses the same mechanism and standards implemented in the last phase. In brief, they are:

- RSA with 4096-bit key pair for public key cryptography
- AES with 256-bit key for symmetric cryptography
- Diffie-Hellman key exchange using the same prime order
- SHA256 for hashing

This documentation goes through each of the new threats and the mechanisms used to protect the system against them.

T5 - Message Reorder, Replay or Modification

Jack

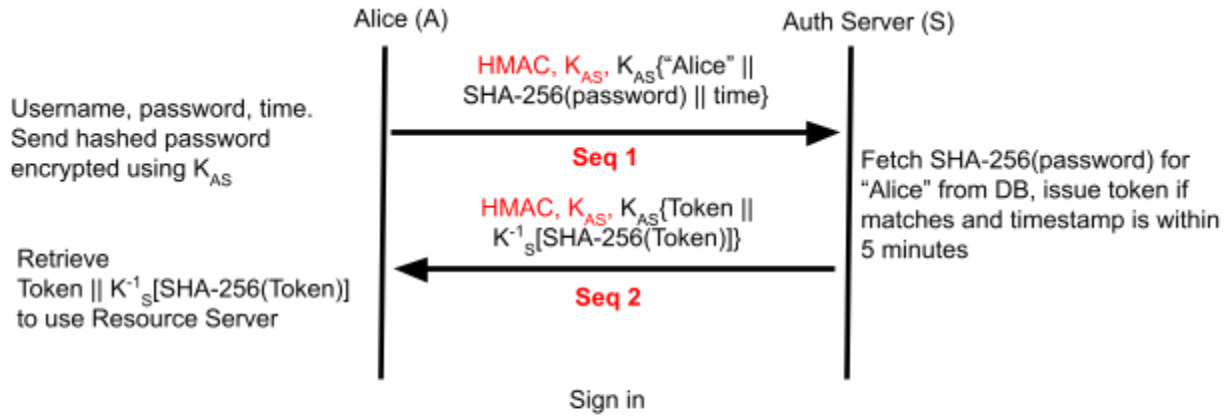
Threat:
Our current security model does not prevent MiTM attacks. A MiTM attack could potentially allow the attacker to fake a client's or a server's identity, and exploit it to gain/leak information. This would be catastrophic.

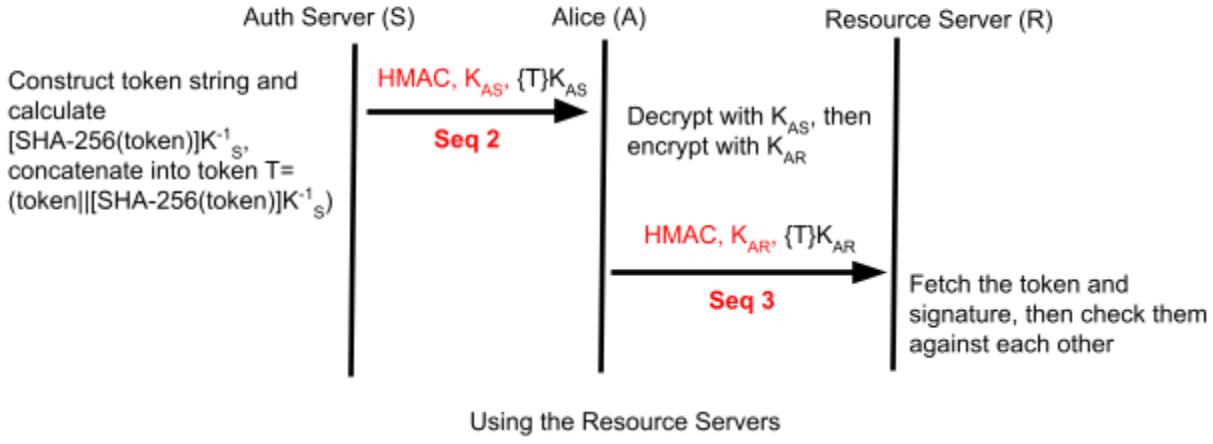
Mechanism:
HMAC is going to be used to verify the integrity of all messages.
Each message is:

1. Shared secret key generated and transported to the receiver (by using our implementation in T4).
2. Generate the ciphertext, which is encrypted by the shared secret key between the two ends.
3. Generate hash (HMAC) of the shared secret key with the ciphertext.

Then send the message:

4. Timestamp (with Seq #) all messages between client and server.
5. If any of the messages are out of order (both client & server checks for this) or the hash is wrong (receiver calculates HMAC and compare), logout the user (delete the token).





Note:
Everything in black is copied from the graphs of T1 & T2, in order, in the last phase. K_{AS} and K_{AR} are derived from public keys of both ends; the shared secret keys are generated and transported as specified in our implementation of T4.

Assumption:
Proper implementation of T4 is required for K_{AS} and K_{AR}.

Justification:
We realized a lot of our protection implementation from the last phase can be utilized here.
Timestamp protects against reorder & replay. It's not possible for Seq 2 to happen before Seq 1 since the token in Seq 1 is needed before Seq 2. Seq 3 cannot happen before Seq 3 since token T is generated at Seq 2.
By using HMAC, we ensure protection against modification of all the messages.

T6 - Private Resource Leakage

Parv

Threat:
The current implementation of the system assumes that the properly authenticated resource servers are trustworthy. This threat introduces the potential for the resource server to be compromised and leak confidential data to a third client. This third client can in turn read and leak private posts, comments and user information posing a breach in confidentiality. The post might be meant for specific users with sensitive information with a possibility to cause emotional or financial damage in multiple ways. With the introduction of this threat, the data in the resource server cannot be stored in a meaningful form, or in a form that could be converted into meaningful form by the resource server or any other client to which the server leaks this information to.

Mechanism:
To solve this problem, the system will have a 256-bit AES symmetric key associated with each post, stored with the auth server. Owing to the implementation in T3 and T4, the data exchanged between a user and each of the servers will happen encrypted by their respective shared key.

1. The post creator will generate the initial 256-bit AES key while creating the post and send the key, an automatically generated unique post ID and the list of users it wants access to the post to the auth server.
2. The auth server maintains this data, along with the post creator's user ID.
3. The user will encrypt the post using this key and send it to store it in the resource server.
4. Only the post creator has rights to add or remove a user from the list of users that have access to the post. When it adds a user to the list, the auth server will

just add this new user to the user list it maintains against the post ID. When it removes a user, the client will also send a newly generated random 256-bit AES key to the auth server. The server will update the user list and store this new key.

5. The post creator will then encrypt the post using this new key and send it to update the post in the resource server.
6. When a user wants to access this post, it will request it from the resource server, and obtain an encrypted form. Then, it will request for the key from the auth server by sending a request with the post ID. The auth server will check if the user list associated with the post ID contains this user, if it does, then it will send the required key to the user.

Note: Our existing system implementation is such that we do not have groups that share access to posts, rather, we have the user define its group when it creates a post. In this, we do not require Leslie Lamport or a mechanism as such to rotate the keys.

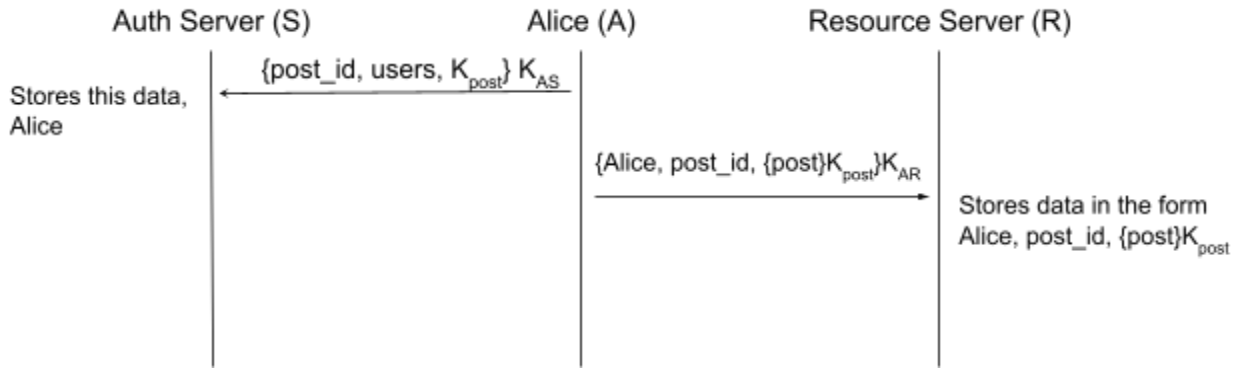


Figure explaining creating a post

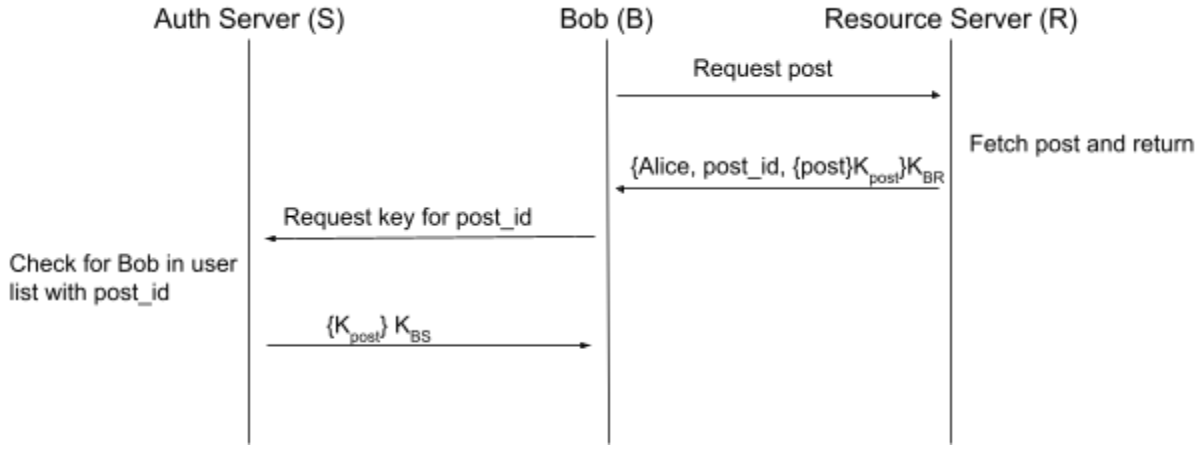


Figure explaining accessing a post

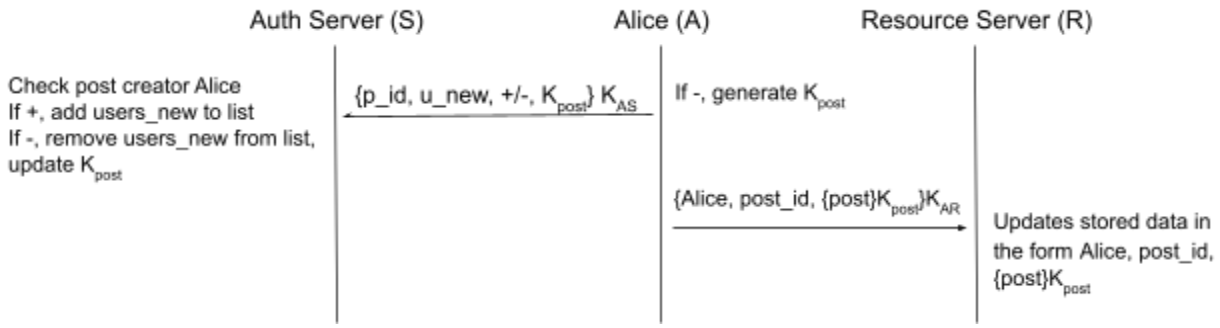


Figure explaining changing post access

Assumptions:
Similar to the previous iterations, we assume that the auth server is trustworthy. We also assume that the resource server does not modify the encrypted value. However, even if it does, because it does not have the required key, it will not have catastrophic impacts. We also assume that in case a user uses an already existing key on purpose, that would be a problem on their part. A random 256-bit key is too large to get "repeated" by chance.

Justification:
This mechanism solves multiple security issues in the system if the resource server is compromised. First, storing data using a shared secret key that the resource server does not have access to prevents the resource server or any malicious user to make sense of the data. Then, with the auth server properly authenticating users, an attacker cannot impersonate another user to get the post's key. This also ensures that only the post creator is able to change the user list that have access to the post. As generating a 256-bit key is very fast and is done on the client side, it will not be exhaustive to create a post for both the client and the auth server. Because every post has a dedicated group of users, we do not require a mechanism like Leslie Lamport where removing a user would require some computation.
Note that in this implementation, we never have the auth server and the resource server talk to each other. Nor is there any "secret" shared between them.

T7 - Token Theft

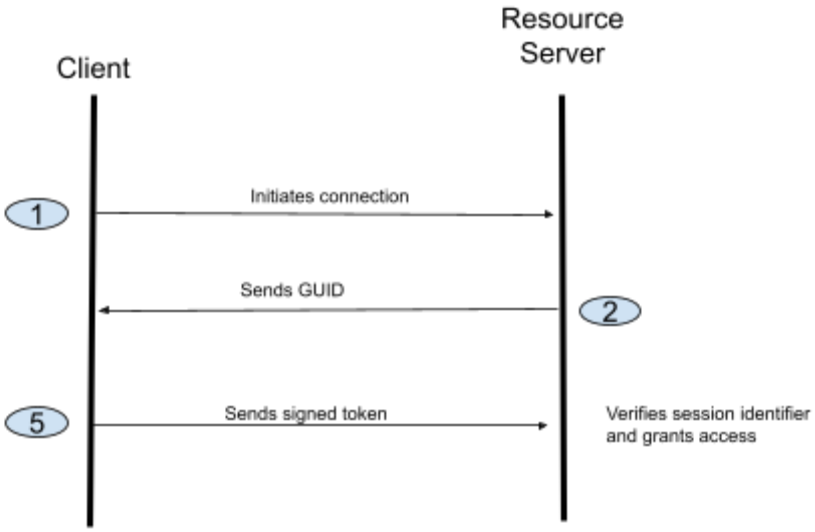
Kadin, Purva

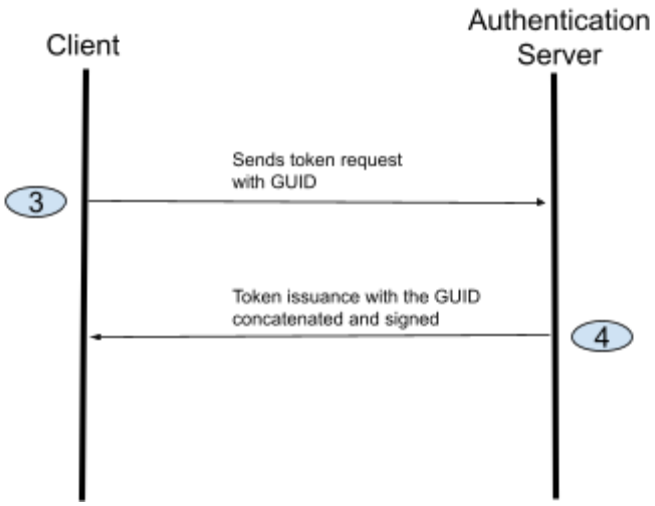
Threat description:
In an environment where resource servers are potentially malicious, they may attempt to steal tokens from clients and pass them to unauthorized users. Such a situation

endangers the system by allowing a user to impersonate another, gaining access to operations and information they are not entitled to. For example, if a user's token is stolen by a compromised resource server and passed to another user, the attacker could gain access to private data, manipulate group memberships, or even compromise the victim's account.

Protection mechanism:
The mechanism to prevent token theft and ensure that tokens cannot be misused if intercepted by a malicious resource server can be outlined as:

1. **GUID as Unique Session Identifier:**
    a. Each resource server generates a GUID, which serves as a unique session identifier.
    b. When a user connects to a resource server, this GUID is shared with the user to establish a session identity.
2. **GUID as Part of Token Request**
    a. Once the client grabs the GUID, it includes this GUID in its token request to the Authentication Server (AS).
    b. The AS then binds the session identifier (the GUID) to the token.
    c. Tokens are additionally generated with an expiration of 2 hours
3. **Session-Specific Token:**
    a. The AS issues a token that is specific to the session, identified by the GUID of the resource server's public key.
    b. When the client receives the token, it sends the token to the resource server as part of its request.
4. **Verification by Resource Server:**
    a. The resource server, upon receiving the token, checks that the session identifier in the token matches its own GUID.
    b. This ensures that the token is intended for use with that specific server and session.





Assumption:
Correct Implementation of T2 ensuring tokens are tamper-proof and that Auth server is trustworthy

Justification:
By using this process, the client ensures it is communicating with the correct resource server by verifying the server's public key hash against a trusted source. The use of a session-specific token, which includes the verified GUID, adds an additional layer of security by binding the token to the particular session and server, thus preventing misuse of the token by other servers or in other sessions. Additionally, a 2 hour expiration date can prevent tokens from being used indefinitely.