Introduction:
Most of the encryption used in this phase revolves around GCM thanks to its secure, fast, and tamper proof properties. This includes T5 and T6. RSA signature is again employed to ensure request integrity during T7.

T5 - Message Reorder, Replay, or Modification

Should this threat happen, the integrity of messages would be put at risk. it would be impossible to tell if the content of the messages is what was originally intended by the sender. As such, it would prove impossible to achieve plausible deniability. A sender could not prove that a message that was maliciously edited wasn't the original message.

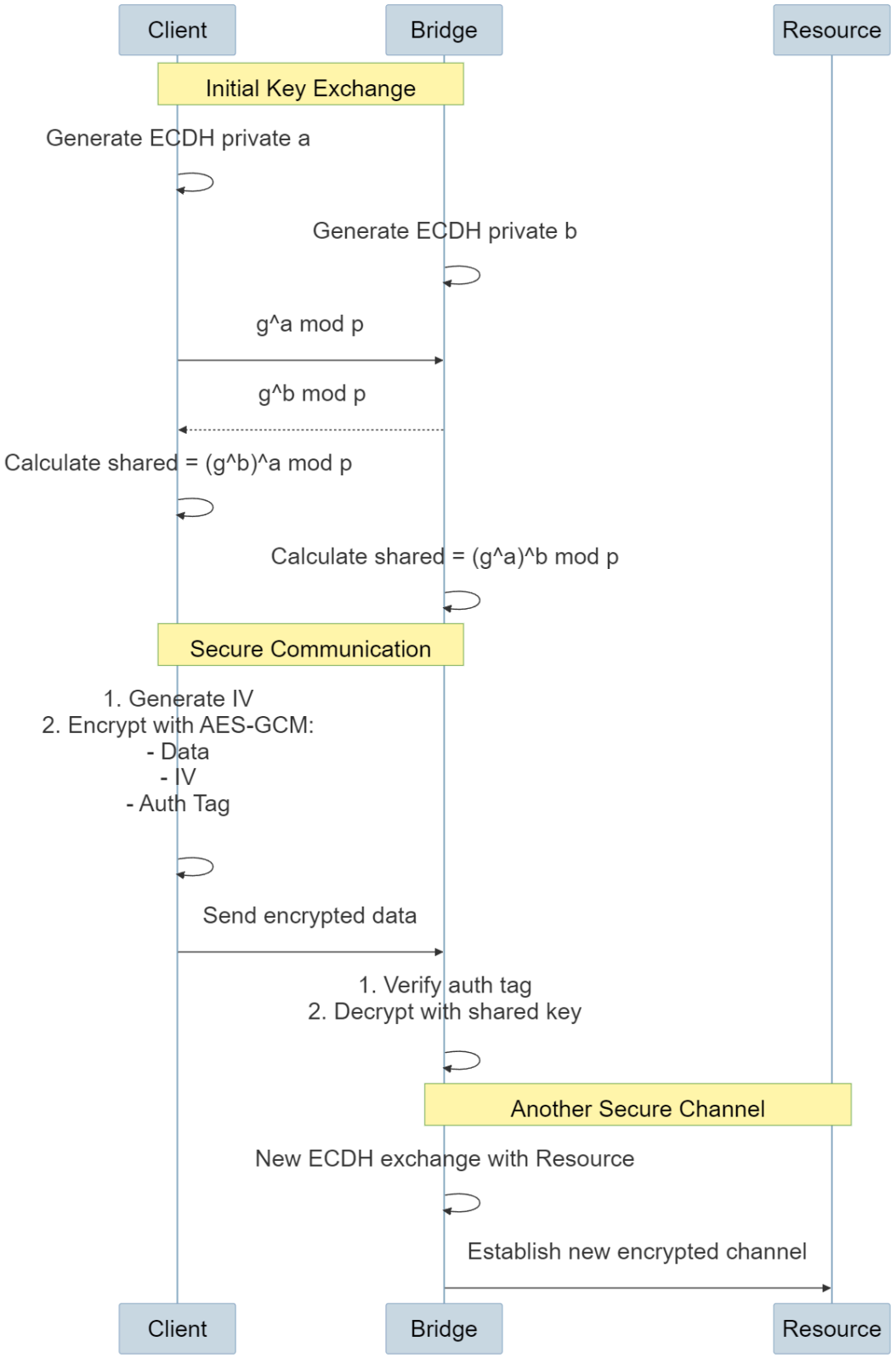
To deal with reordering and replay, all messages will now also contain sequence numbers. Each party (Sender, receiver) will maintain a counter starting at 0. As subsequent messages are sent, their sequence number will increase by 1, therefore determining the expected order of the messages. To ensure that the counters themselves remain secure, they themselves could be encrypted with the same GCM key that is used during T4. Given our design with a per-user session, message order can be stored on a per-user basis. This measure effectively protects against message reordering or replay, since any attempts will result in a mismatching sequence number, and the connection will be terminated.

Finally, to deal with modification, we believe that our AES-GCM implementation already provides robust protection against this. GCM contains an authentication tag, which effectively acts as a way to ensure that no modification to the ciphertext or the IV has happened.

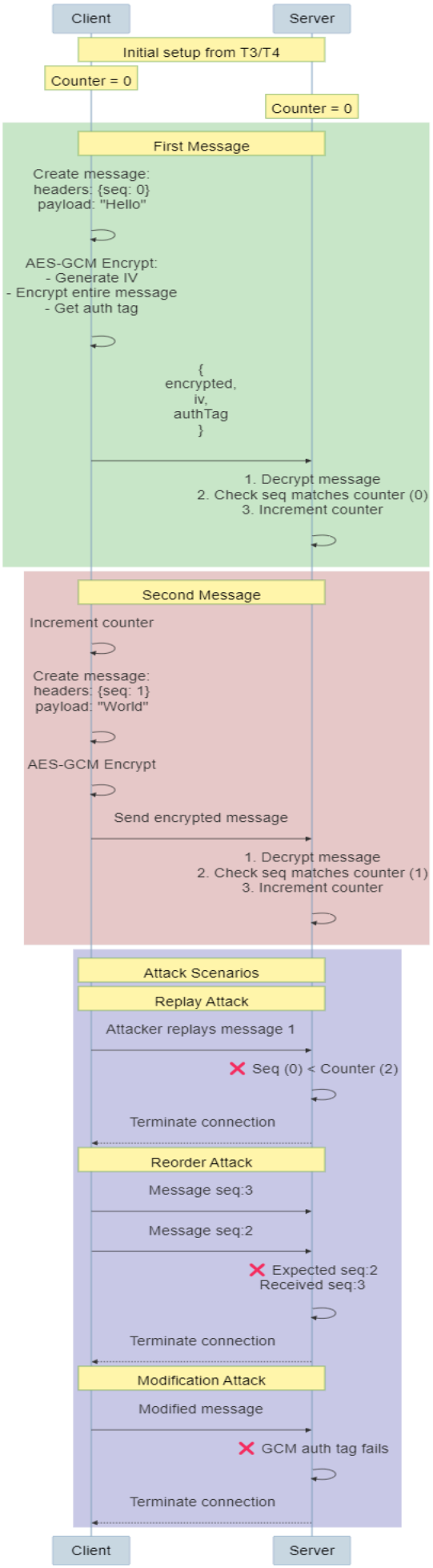
Here's how the message would be structured:

```
{
        Headers: {
        seq:1,
        },
        payload: "message"
}
```

Keep in mind, both the payload and sequence are encrypted.
Recall our T4 secure channel handshake:



Which will now have the following addition to deal with the active attacker:


Client
Server
Initial setup from T3/T4
Counter = 0
Counter = 0
First Message
Create message:
headers: {seq: 0}
payload: "Hello"
AES-GCM Encrypt:
- Generate IV
- Encrypt entire message
- Get auth tag
{
encrypted,
iv,
authTag
}
1. Decrypt message
2. Check seq matches counter (0)
3. Increment counter
Second Message
Increment counter
Create message:
headers: {seq: 1}
payload: "World"
AES-GCM Encrypt
Send encrypted message
1. Decrypt message
2. Check seq matches counter (1)
3. Increment counter
Attack Scenarios
Replay Attack
Attacker replays message 1
Seq (0) < Counter (2)
Terminate connection
Reorder Attack
Message seq:3
Message seq:2
Expected seq:2
Received seq:3
Terminate connection
Modification Attack
Modified message
GCM auth tag fails
Terminate connection
Client
Server

T6 - Private Resource Leakage

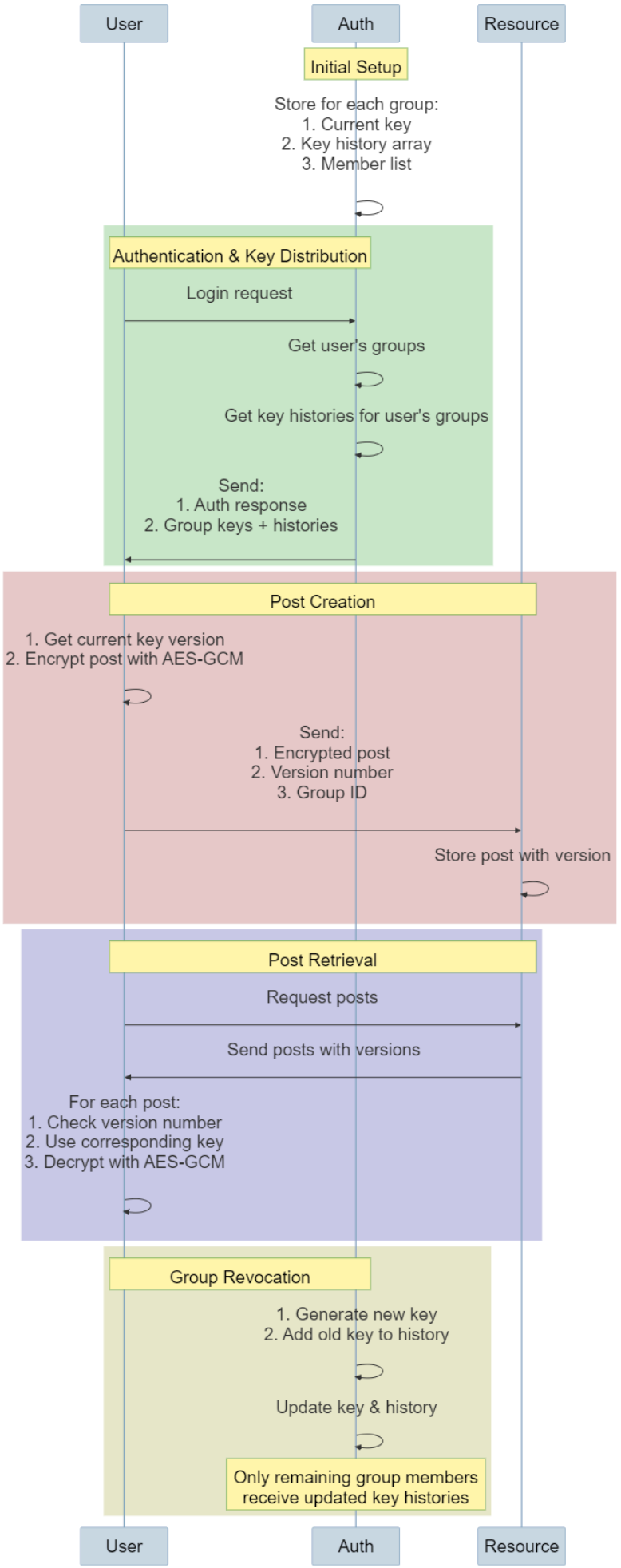
This threat and its consequences are quite simple. A rogue resource server could be able to leak private resources to entities without the required permission levels, making it so that secrecy is compromised.

The first step to deal with this is to store the messages in an encrypted manner, such that the resource server is not actually capable of reading them, being limited to just distribution. For the encryption method, we believe that GCM works great again since it provides confidentiality, and authenticity through the aforementioned authentication tag, not to mention its programmatic efficiency.

Now, the keys to carry out message decryption would be stored on the authentication server. This means that the auth server now needs a new addition to the database to keep track of each key for each group. Recall that the auth server already has stored the groups that exist, and which user belongs to which group. As such, on successful authentication, the keys necessary for message decryption (for the groups that the user has access to) will be sent in addition to the existing response.

In the event of group revocation, a form of lazy re-encryption will take place. In practical terms this means that the key to that group the user was removed from will be updated. This will prevent the removed user from continuing to access new posts on that group. We now have to deal with posts that were encrypted with the previous key. To do this, we'll keep a complete history of all the group keys that have been used for each group, and send this history to the clients that remain in this group. This way, all files can be decrypted. This of course means that in theory a user that has been removed from a group could still see the posts in that group prior to his removal. We are ok with this, since he would've been able to see them anyway during his time in the group. To know which key decrypts which post, a version structure will be implemented. This version will be appended to each post once created and encrypted. The version number is determined by how big the array containing the key history is at the time of post creation. The resource server will therefore have a new field for each post detailing its version number, that it will send alongside the encrypted post itself.

Additional details: the latest version is tracked by the AS. Keys are sent in an object format like so: {00:'sfsdfsfsdfojsd', 01:'odasfdsdfosdf', etc). Each number indicates the version, so the client simply finds the biggest number and it knows that is the current key that should be used: `const latestVersion = Math.max(...Object.keys(groupKeys).map(Number));`
This way it also knows what version it is on, so this version value is appended to all posts. This is forwarded to the RS when a resource is uploaded. The RS stores this alongside the rest of the post information like title, user, etc. This is included in the response once posts are requested, with every post having its own version value.


User
Auth
Resource
Initial Setup
Store for each group:
1. Current key
2. Key history array
3. Member list
Authentication & Key Distribution
Login request
Get user's groups
Get key histories for user's groups
Send:
1. Auth response
2. Group keys + histories
Post Creation
1. Get current key version
2. Encrypt post with AES-GCM
Send:
1. Encrypted post
2. Version number
3. Group ID
Store post with version
Post Retrieval
Request posts
Send posts with versions
For each post:
1. Check version number
2. Use corresponding key
3. Decrypt with AES-GCM
Group Revocation
1. Generate new key
2. Add old key to history
Update key & history
Only remaining group members receive updated key histories
User
Auth
Resource

T7 - Token Theft

In this scenario, a malicious server could use the token to grant the client access to resources that they normally wouldn't be allowed to access, again compromising the security of the data and allowing for direct session hijacking.

To solve this, we'll add unique identifiers to the requests to each server, so that they know the tokens sent to them are legitimate. Recall that this project uses per-user sessions that are formed once the response by the authentication server is receded. We'll adapt this process in the following way:

Before, every request to the resource server was accompanied by a signed AS object with all the necessary details for correct post processing:

{
isAuthenticated: boolean,
isVIP: boolean,
isAdmin: boolean,
username: string,
groups: list,
userID: integer
}

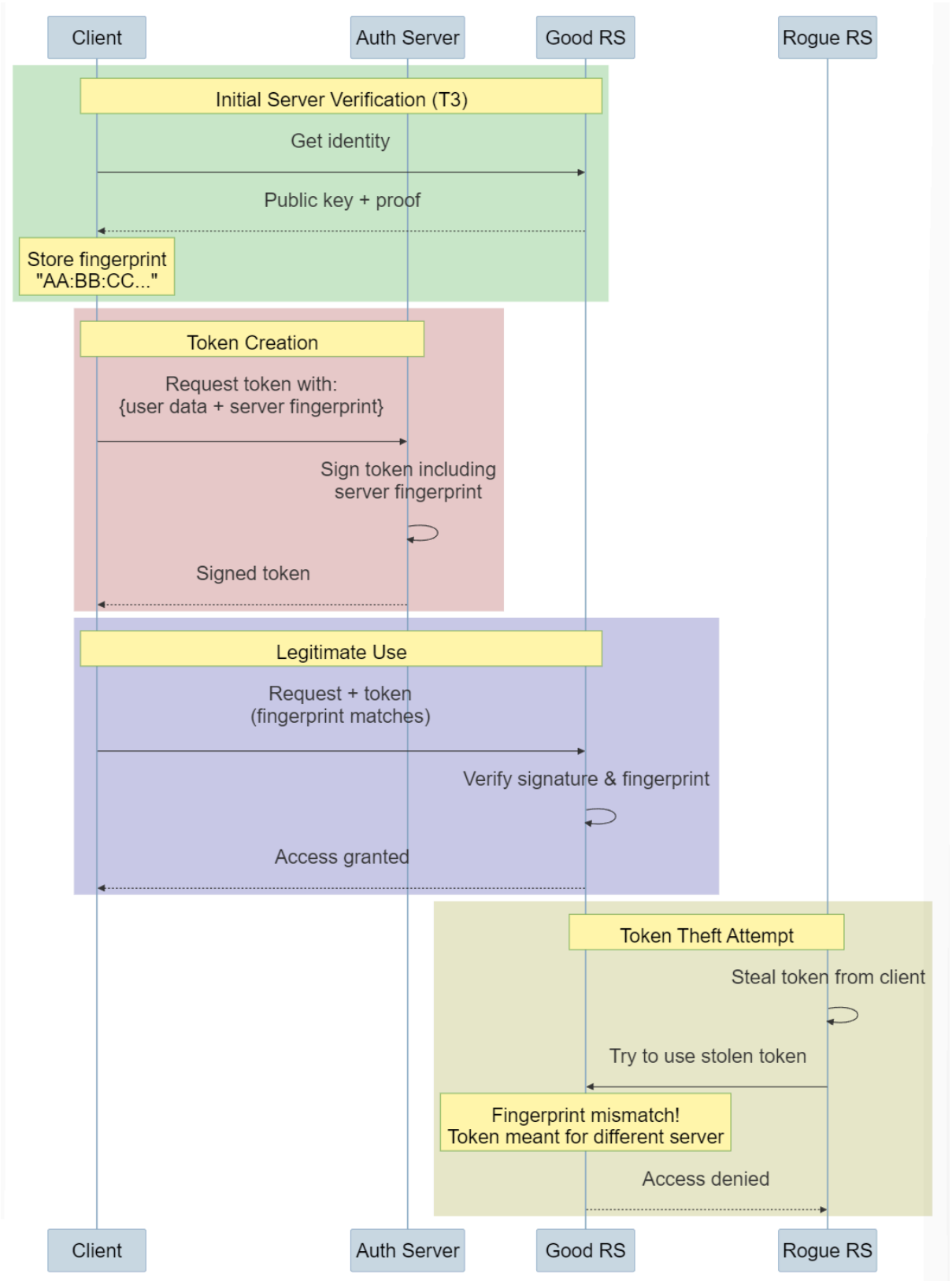
Now, this object will have an additional field, the server fingerprint:

{
isAuthenticated: boolean,
isVIP: boolean,
isAdmin: boolean,
username: string,
groups: list,
userID: integer
serverFingerprint: "AA:BB:CC..."
}

Where this field is decided by the user when he decides what resource server to connect to. This new object is then sent to the AS for it to sign, and sent (signed) to the intended destination RS. When the contents of one of the resource servers are needed, only a token that matches its fingerprint will be sent. Therefore, even if the rogue server attempts to steal this token and use it on another resource server, it will detect that the received fingerprint field does not match its own, and close the connection. Since this object is RSA signed, it is not possible for the rogue RS to replicate it with another server's fingerprint.

FINGERPRINT CALCULATION: Done by hashing the RS public key.

(This fingerprint is the one that was acquired during the T3 server verification exchange). Below is a visualizer:


Client
Auth Server
Good RS
Rogue RS
Initial Server Verification (T3)
Get identity
Public key + proof
Store fingerprint
"AA:BB:CC..."
Token Creation
Request token with:
{user data + server fingerprint}
Sign token including
server fingerprint
Signed token
Legitimate Use
Request + token
(fingerprint matches)
Verify signature & fingerprint
Access granted
Token Theft Attempt
Steal token from client
Try to use stolen token
Fingerprint mismatch!
Token meant for different server
Access denied
Client
Auth Server
Good RS
Rogue RS

This part proved to be outstandingly challenging given our erroneous preconceptions of the desired project structure. We had to adapt our solutions, especially in T7. GCM proved so effective during the last phase, that the one unifying factor in most of our protocols is the full use of this method. All of the solutions proposed during the last phase remain just as secure. There was no fundamental alteration to the way T1-T4 are handled, and instead our current solutions are built on top of this. This stepping stone approach allows us to design and implement new features, while still respecting the existing ones.