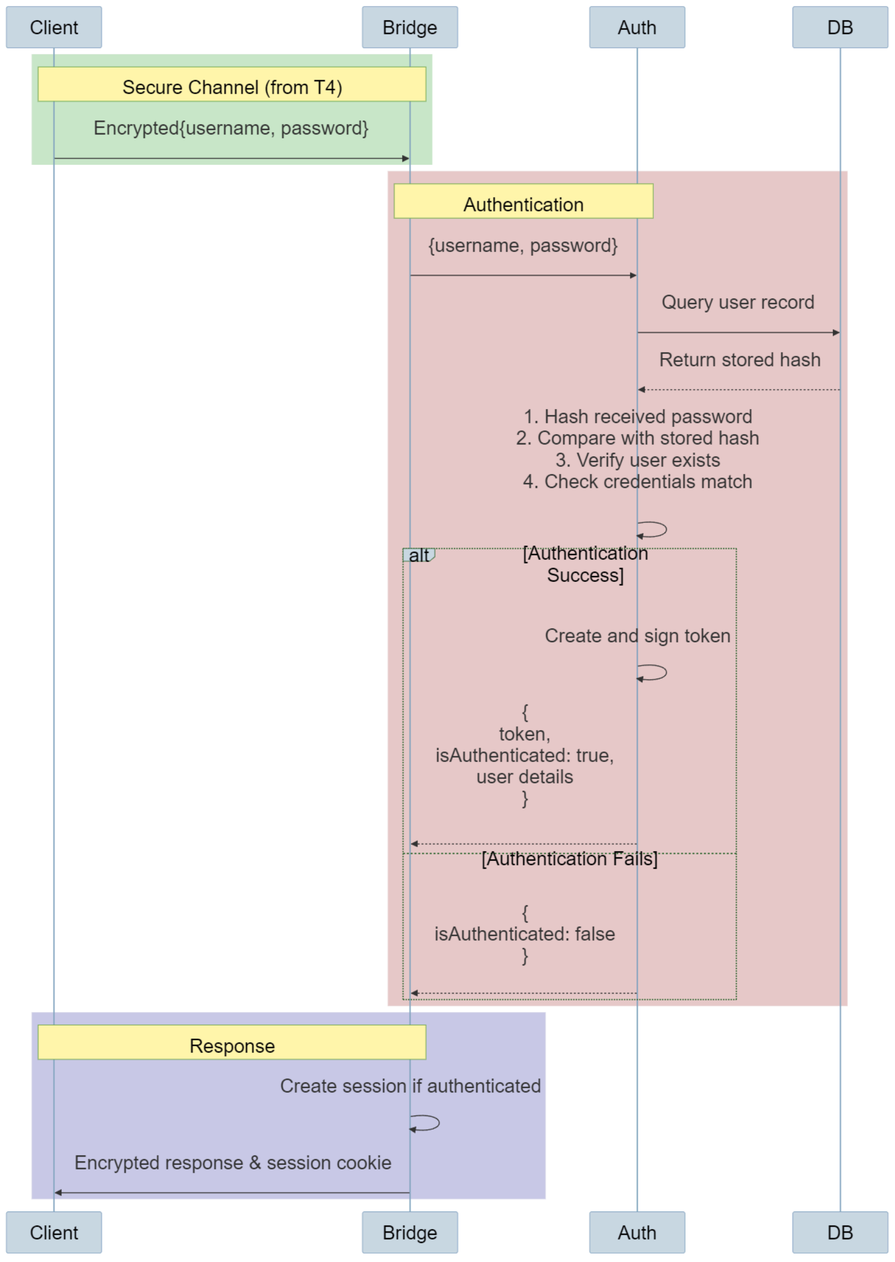
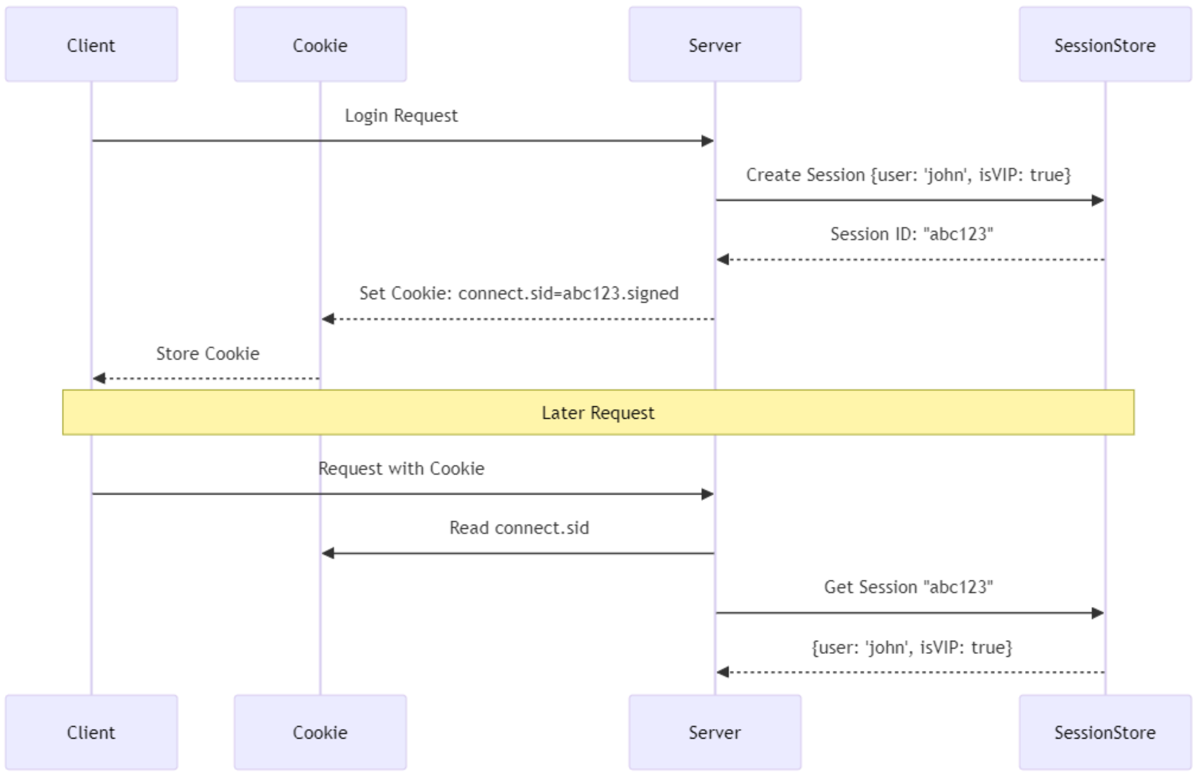
T1 Unauthorized Token Issuance

The main threat of this section is stealing another person's account. If an attacker successfully did this, they would be able to view, post, or delete posts on behalf of the original owner, compromising the security of the messages. Applied more closely to the project, a reasonable assumption for the motivation for this attack would be the users looking to know what's being posted in groups that are not their own, or perhaps gaining access to the vip posts. Recall the basic project information flow, Where a session is created via express-session middleware once the authentication server provides successful authentication. Here's how the authentication happens after all proposed changes:


Client
Bridge
Auth
DB
Secure Channel (from T4)
Encrypted{username, password}
Authentication
{username, password}
Query user record
Return stored hash
1. Hash received password
2. Compare with stored hash
3. Verify user exists
4. Check credentials match
alt
[Authentication Success]
Create and sign token
{ token, isAuthenticated: true, user details }
[Authentication Fails]
{ isAuthenticated: false }
Response
Create session if authenticated
Encrypted response & session cookie
Client
Bridge
Auth
DB

On successful authentication (checking the database for the user records), a session object containing all the details of the user is created, including: Admin status, Groups, VIP status, Username, and the isAuthenticated boolean that is only true if the Auth server determines so. It is important to add that sessions are **only created once the Auth server has successfully responded**. As for user registration, users may be able to register themselves, or an admin may create a user. If a user registers, they won't have access to

any resource until the administrator has added them to a group, so an attacker won't be able to register to access any information since he doesn't have the metadata to do so.
The main threat is for a malicious user to hijack this session object after its creation. A crucial part in securing the project is to make sure that the session can only be accessed by the legitimate user. This is achieved via a cookie system, where the session is tied to a cookie stored in the client:



The cookie contains unique data that ties it directly to its session, making it infeasible for an attacker to manipulate it to access another session. Here's an example of what the cookie would look like:
connect.sid=s%3Aabc123def456.RSc0jHJBK4YQ4xN%2BHoKomcwNw4Y where:
's%3A' is a prefix indicating this is a signed cookie
'abc123def456' is the actual session ID
'.' is a separator
'RSc0jHJBK4YQ4x...' is the signature created using the secret
The session id is tied to the signature. Even if an attacker wanted to hijack the session with another token with the same session id, it would have the incorrect signature.

The final part is to make sure a malicious user cannot access the usernames and passwords from other users. Currently, this information is stored in a non encrypted sqlite database. If a malicious user got a hold of this data, there would be nothing to stop him from authenticating, and creating a legitimate session token allowing him to perform all operations allowed for the original user. Therefore, **user password information will be encrypted by the Auth server via Bcrypt.** It's made specifically for password hashing, it's got built-in salt generation, and it can potentially be made slower to prevent brute force. This last factor is of special interest for this project, since currently there's no mechanism to prevent repeated login attempts.

T2: Token Modification/Forgery

This project works in the following way: After authentication is complete, the authentication server sends a JSON object to the bridge server. It has the following structure:

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

The bridge then takes this object, copies it, and stores it on a per user basis.

```
if (responseObj.isAuthenticated) {
            req.session.resourceServerObj = {
                isAuthenticated: true,
                isVIP: responseObj.isVIP || false,
                isAdmin: responseObj.isAdmin || false,
                username: receivedParams.username,
                groups: responseObj.groups,
                userID: responseObj.userID
            };
```

(responseObj is the response from the authentication server)
This copied object, stored in the node.js bridge is the "Session". It is sent on every request to the resource server, so that it knows what user is attempting to read/post/delete content, and if they have permission to do so.

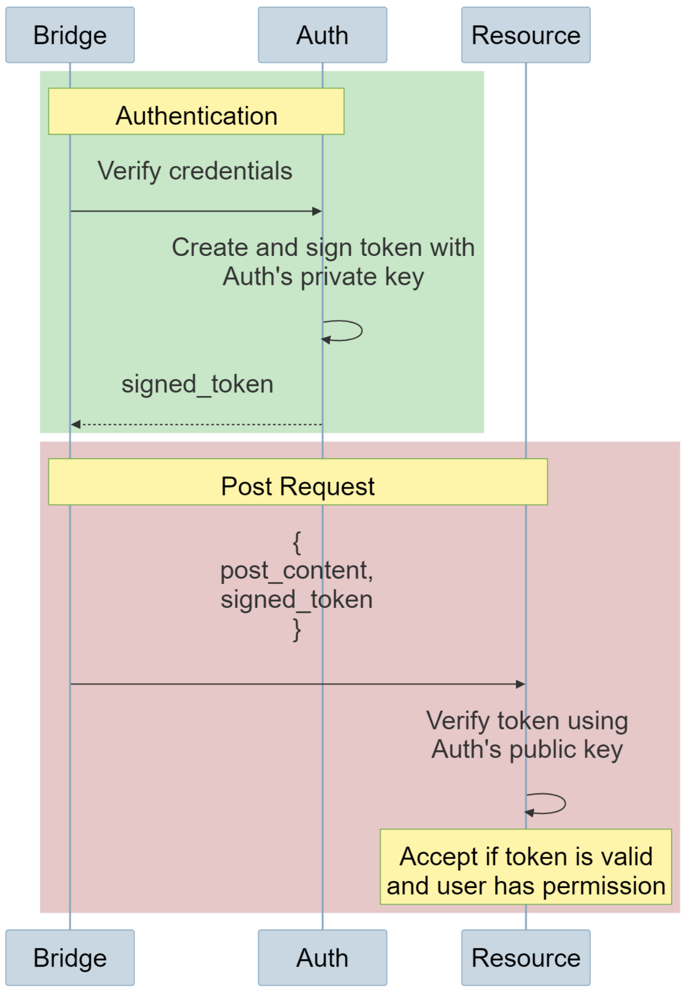
This session object is what is accessed by the cookie.The reason that it's important that the cookie is signed is that no other client can access this session object. Each client has its own separate session object with their individual metadata. Each individual session has its own id, which must be matched by the session id in the cookie.
This object is what should be considered as the token.

In order to assure authenticity of the token, we propose that the authentication server adds an RSA signature to the initial JSON object sent to the bridge server.

Firstly, since the session is what controls user details and permissions, the session could be considered to be the token itself. As a reminder, the session was made using the response from the authentication server. The ability to alter or forge cookies to access other sessions would effectively compromise the token.

Therefore, cookies must be protected from forgery. Thankfully, our implementation allows us to avoid this problem. You might recall that we previously mentioned a "secret". This is a parameter made during the session configuration. It is a randomly generated string that effectively acts as the key. This specific middleware uses the hashing algorithm

HMAC-SHA256 combined with the secret key to generate the signature. As shown above, the signature is appended to the cookie and verified on every subsequent request from the client.
Now, the session itself also contains a signature given by the auth server which is checked by the resource server. Initially, the auth server has generated a 2048 bit RSA key pair. After the authentication process is complete, the auth server signs the response to the bridge with its private key (The resource server is configured with the auth server's public key). Therefore, all requests to the resource server contain this signed token, and this signature is checked to make sure the request is legitimate.


Bridge
Auth
Resource
Authentication
Verify credentials
Create and sign token with Auth's private key
signed_token
Post Request
{ post_content, signed_token }
Verify token using Auth's public key
Accept if token is valid and user has permission
Bridge
Auth
Resource

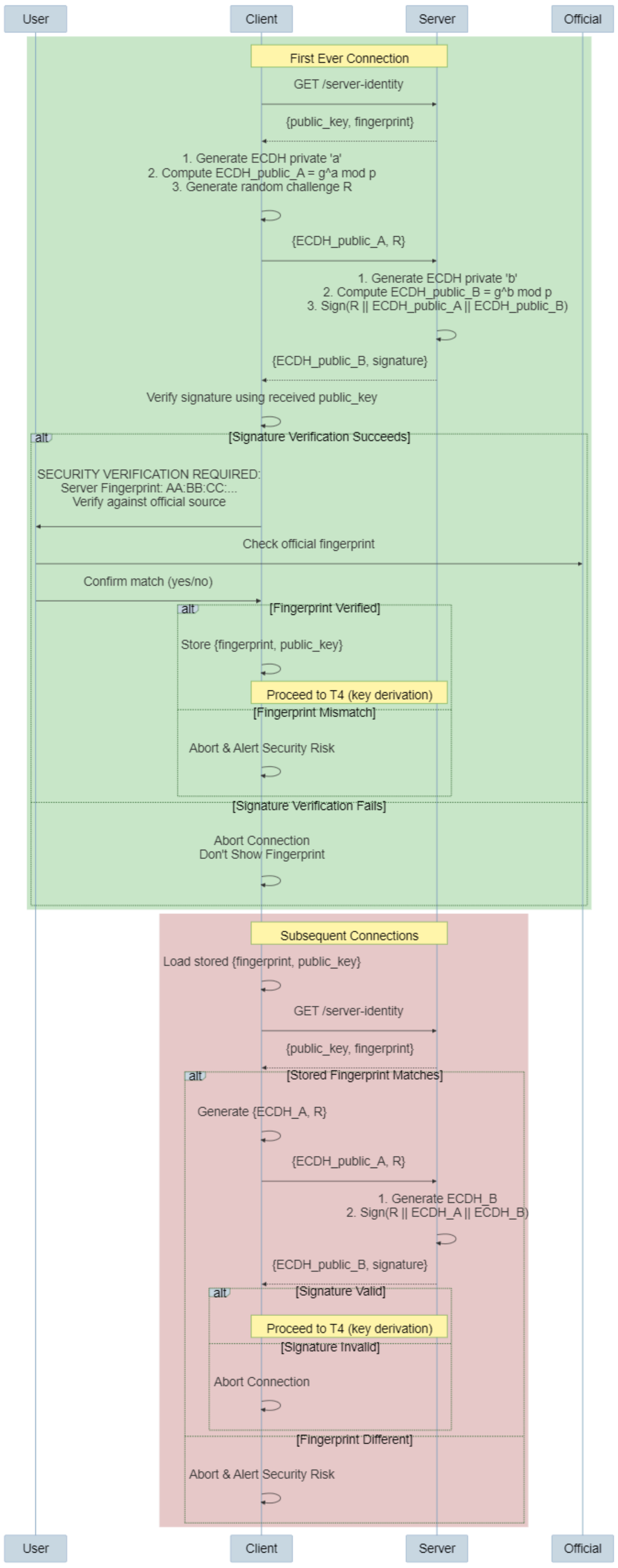
Why this works: The attacker could try to 1) Alter the cookie to hijack a session or 2) Try to create a fake session to access the resource server. 1 is solved thanks to the cryptographically signed cookie previously mentioned, while 2 is solved thanks to the fact that the session has been signed by the authentication server, so any requests to the resource server that do not have this signature are blocked.

T3: Unauthorized Resource Servers

Since the resource server stores all posts, an unauthorized resource server would compromise the integrity of the data, sharing all posts with a malicious third party and perhaps even storing them with no chance to actually delete them. There would be no way for the client to notice that they're communicating with the wrong server. Therefore, a

verification of the server identity must be carried out. **NOTE**: the structure of this project entails that the client does not actually communicate with the resource server, since all communications go through the bridge server instead. Therefore, the verification process that will be presented below will apply to BOTH bridge server and resource server, in an effort to prevent malicious bridge server impersonations. **The bridge authenticates itself to the client, and the resource server authenticates itself to the bridge.** Assume all steps pertaining to “the server” include both.
For our solution, the client and the server will generate a 2048 bit RSA key pair, the private key is kept secure on a server and the public key is distributed to the client (React client or bridge server in this case). The server also calculates a fingerprint, parsing the public key through a SHA256 hashing function. Finally, the fingerprint is distributed through a secure channel, most likely official documentation. Now, the steps differ on first and subsequent connections:

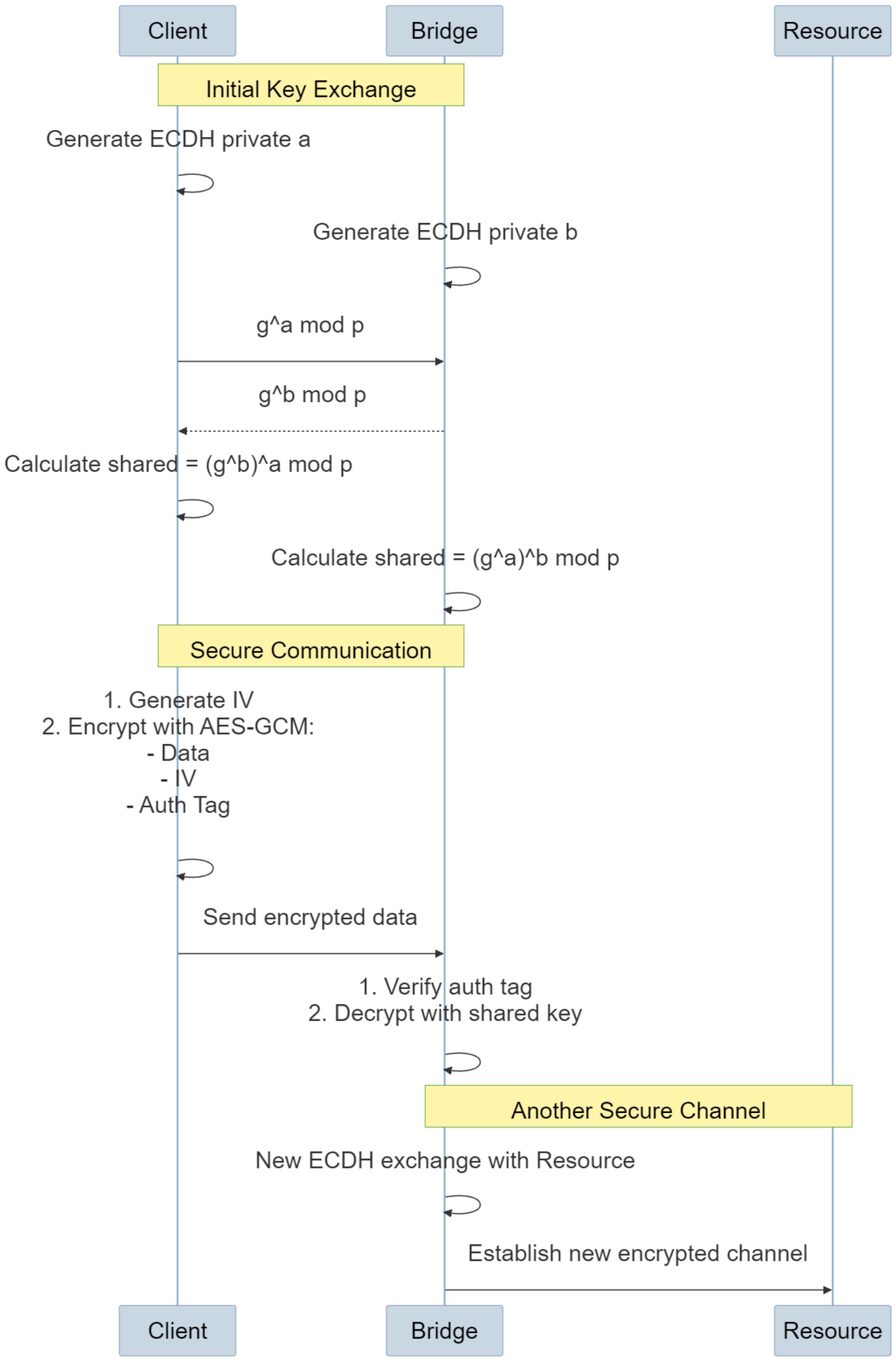
First connection:
The client will call a get endpoint designated to identify the server. The server responds with its public key. The client derives the fingerprint from this. The client then generates a random ECDH private key a, and computes the public value A = g^a mod p. The client then creates a challenge R. Both ECDH public A and challenge R are sent to the server.
The server then generates a random ECDH private key b, and computes the public value B = g^b mod p. The server then concatenates R with ECDH public A and ECDH public B. It then signs this concatenated string using its private key (RSA) and sends it back alongside ECDH public B to the client. The client then verifies the signature, and if successful, proceeds to user verification.
User verification occurs as follows: The following message is displayed to the user: "CRITICAL SECURITY VERIFICATION Server has proven possession of private key. Server Fingerprint: AA:BB:CC:... You MUST verify this matches the official fingerprint from: Official documentation: [URL] This fingerprint will be used to verify the server's identity in all future connections. WARNING: Proceeding without verification risks security breach Does this fingerprint match the official source? (yes/no)" **Given that this application is designed for a low scale small user count, where there is manual administration of every user’s permission levels, it is realistic to imagine in this context that the secure link containing the expected fingerprint is sent out by the administrator to the users, ensuring that the expected fingerprint has not been forged.**
Should the user confirm the match, the client will securely store the server’s fingerprint. (ECDH exchange would follow here, which will be explored in detail during T4). Keep in mind the ECDH key exchange is bound to the verification through the signature sign(R || ECDH_A || ECDH_B).
Subsequent connections:
During subsequent connections, no user interaction is needed, and fingerprint verification is done automatically using the stored trust data. All other steps play out the same way. Below is a visualization:


User
Client
Server
Official
First Ever Connection
GET /server-identity
{public_key, fingerprint}
1. Generate ECDH private 'a'
2. Compute ECDH_public_A = g^a mod p
3. Generate random challenge R
{ECDH_public_A, R}
1. Generate ECDH private 'b'
2. Compute ECDH_public_B = g^b mod p
3. Sign(R || ECDH_public_A || ECDH_public_B)
{ECDH_public_B, signature}
Verify signature using received public_key
alt
[Signature Verification Succeeds]
SECURITY VERIFICATION REQUIRED:
Server Fingerprint: AA:BB:CC:...
Verify against official source
Check official fingerprint
Confirm match (yes/no)
alt
[Fingerprint Verified]
Store {fingerprint, public_key}
Proceed to T4 (key derivation)
[Fingerprint Mismatch]
Abort & Alert Security Risk
[Signature Verification Fails]
Abort Connection
Don't Show Fingerprint
Subsequent Connections
Load stored {fingerprint, public_key}
GET /server-identity
{public_key, fingerprint}
alt
[Stored Fingerprint Matches]
Generate {ECDH_A, R}
{ECDH_public_A, R}
1. Generate ECDH_B
2. Sign(R || ECDH_A || ECDH_B)
{ECDH_public_B, signature}
alt
[Signature Valid]
Proceed to T4 (key derivation)
[Signature Invalid]
Abort Connection
[Fingerprint Different]
Abort & Alert Security Risk
User
Client
Server
Official

Why this works: This approach is designed to prevent session hijacking. The server's signature includes exactly which ECDH values are being used, so the attacker cannot change ECDH_A to ECDH_X since the signature won't verify, use a signature with different DH values, or create their own signature since they don't have the private key. Since the ECDH values must be respected, hijacking is not possible. 2048 bit RSA is commonly used and provides strong security for most applications. Finally, the fingerprint acts as a checksum for identity verification, ensuring the public key received is the one that's expected.

T4: Information Leakage via Passive Monitoring
The current state of the project allows for unencrypted communication, meaning that plaintext is sent across client and servers for all checks and processes. In the presence of a passive listener, this proves to be an unacceptable risk since both user data such as usernames and password, and the actual post content would be available to see. End-to-end encryption must be applied. Again, both communication between client and bridge, and communication between bridge and authentication/resource must be encrypted. This is in order to ensure information security at all points. Elliptic Curve Diffie hellman will be used at the initial communication phase to establish a secure channel, then gcm will be used to encrypt and securely send information from there on. This two stage system is used since ECDH provides a secure way to establish a shared key, but is much too slow for subsequent communications. GCM is chosen over other popular modes since it offers excellent performance while being generally secure and reliable. **NOTE:** P-256 curve (256-bit) is used for ECDH. It is standardized and well analyzed, has no known vulnerabilities, and is a good balance between security and performance. Recall that ECDH is integrated into T3, and as such part of the process below would've happened previously. We do feel however that it's best to provide the full ECDH protocol for the sake of clarity.

Secure channel establishment with ECDH:
Client generates a random private key a, computes the public value $A = g^a \bmod p$, and sends A to the server. The server then generates a random private key b, computes the public value $B = g^b \bmod p$, and sends B to the client. Both client and server compute the same shared secret: $g^{(ab)} \bmod p$, and derive the AES key.
AES key derivation is done using HKDF, with the input being the shared secret in raw bytes. For salt, we don't use any value since there is no need for added randomization. The AES key size will be 32 bytes.
Message encryption:
(The sender and receiver could be any of the servers in the project or the client).
The sender generates a random IV that is 16 bytes long and never reused, sets up AES-GCM with the shared key and IV, encrypts the message: C = AES-GCM(key, IV, message), gets the authentication tag, and sends the encrypted message along the IV and authTag.
The receiver decodes the IV, ciphertext, and auth tag, sets up AES-GCM with the shared key and IV, verifies the auth tag, and proceeds to decrypt the message. Below is a visualization:



WHY THIS WORKS: ECDH works perfectly in this case since no pre-shared secrets are needed, it’s especially secure, and each session gets its unique key. GCM is a very fast option that can easily keep up with this project’s information flow, but still maintains cryptographic integrity. We achieve perfect secrecy, efficient encryption, and message integrity.

We believe that the combination of the mechanisms presented create a secure application for this phase. Session management through the bridge was an idea that was concocted thanks to the feedback given in phase one. All members had to significantly broaden their horizons to make all parts of the project work, but the session-based design provided a unified testing ground that allowed the project to remain coherent even with each member working on a separate part. Concerning the specific P3 details, the cookie hash signature was discussed and agreed upon as an adequate solution. There was also a general agreement on the necessity to encrypt the databases. GCM was proposed by one of the members, and after a short discussion was also agreed upon.