Group Name: securecoders

Members:
Gavin Abramowitz
Luke Charlesworth
Michael Bartlett
Owen Wurst

P4 Writeup

General Introduction

In order to mitigate the threats of message reorder, replay and modification, private resource leakage, and token theft under the new threat model, our plan is to implement a three part solution: 1) Timestamp, record and, HMAC communications between the client and servers, 2) Only store files encrypted with “group keys”, 3) Generate server IDs to validate tokens for a single resource server at a time. By securing all communications between the client and servers we can prevent active attackers from tapering with our system in a meaningful way. With timestamps and HMACs, replay attacks and message modification can be prevented. Additionally, by maintaining a session state for each user, message reordering during the handshake is protected against. Because the leakage of private resources is inevitable, the content in those resources will be protected because only encrypted files will be stored on resource servers. These files can only be decrypted by the client with their shared group key, ensuring that even if information is leaked, it will be unreadable. Finally, by validating tokens for a single resource server at a time we can stop stolen tokens from being used on any server except the server it is verified for, thus making stolen tokens useless.

Technical Choices

1. Public and Private Key Pairs will be generated with RSA using 4096 bit keys.
    a. **Why:** This is the largest RSA keypairs provided, we chose to sacrifice some computational efficiency in order to give our application a high degree of key security; making brute force key acquisition attacks infeasible.
2. RSA encryption will utilize OAEP padding.
    a. **Why:** OAEP padding is semantically secure and resistant against chosen ciphertext attacks, making it more secure than other options.
3. RSA signing will utilize PSS#1 padding.
    a. **Why:** This padding mode is extremely robust, guarantees a consistent length, and protects against vulnerabilities seen in deterministic padding modes

4. 192-bit AES will be used for encryption.
    a. **Why:** 192-bit is chosen as a middle ground between efficiency and security, as it is the middle key size that AES encryption standards offer. Since these are shared session keys meant to be utilized in many more communications then the RSA keypairs, we wanted to maintain strong security while meeting user demands.
5. CTR mode will be used as the mode of operation with 128-bit block sizes
    a. **Why:** Because AES only supports 128-bit block sizes, this is the block size that will be used. Additionally, IVs will not be reused because IV reuse is not safe.
6. SHA256 for hashing
    a. **Why:** SHA256 is a hash algorithm that is extremely secure. Its properties cause collisions to be extremely unlikely, allowing us to trust the integrity of the function.
7. 6 second Time Epoch for TimeStamp check
    a. **Why:** This is a short enough duration to ensure no attacker could passively monitor and replay a communication exchange, but long enough to account for added computation time between our client and servers.
8. How we perform Out-Of-Band Key Exchange
    a. **Why:** When public keys are exchanged out of band, they will be stored in a txt file that will be passed in as an argument at runtime
    b. **Why:** The auth server will be run first and its public key will be supplied to the Resource Server which will be run second, the Resource Server's public key will be supplied to the client which will be run third.
9. Sha256 (with Java Crypto package) for generating HMACs'.
    a. Sha256 possesses the three properties of secure hash functions, which are essential for ensuring the integrity of communications.
10. JSON web tokens (jwt) signed by the auth server's private RSA key are used for our auth tokens and set with the following claims:
    a. "expires": tokens will be set to expire 10 minutes from when they are issued, the auth server has an endpoint to refresh a token, resetting the expiration for 10 minutes from the time of refresh, that can be used if the token has not yet expired
    b. "uid": a user id is in the token so that the verifying server knows that the token was generated for this user
    c. "teams": this specifies which resource groups a user is part of so the resource server knows not to allow access to the resources of a group that the user is not a part of
    d. "sessionID": this binds the chain of tokens for a given user (as tokens are likely to be refreshed multiple times during a session) to one session. The resource servers check that this session id is correct so that the handshake protocol does not need to be redone with every token

e. “resourceID”: this binds a token to a single resource server so that the token for one resource server cannot be used for another resource server: this is discussed further in T7

T5 Message Reorder, Replay, or Modification

Introduction to Threat

The threats of message reordering, replay, or modification by an adversary in client-server communications are quite dangerous. Message reordering by an attacker can lead to data dependency issues (as some communications rely on data received previously), transaction tampering, and denial of service (as attackers may reorder messages to disrupt normal functioning of a system). Further, message replay attacks can lead to unauthorized access control as attackers can replay old authentication messages of a user to bypass authentication. Last, message modification compromises data integrity of client-server communications and potential confidentiality risks. These sub-threats all share the common theme of an attacker tampering with a communication between client and server for their benefit.

Examples of an attack based on each sub-threat

Replay:

- User logs into online banking and the bank server sends a unique session token
- Adversary intercepts the communication, capturing the session token
- Once the user logs out, the attacker then replays the captured token to the banks server
- The server sees that this is a valid token, so it grants this access request (unauthorized access), assuming it's legitimate.
- The attacker now grabs the user’s online banking credentials and uses them to pay off their college loans.

Reordering:

- Server expects message parts x, y, and z to be received in sequence for a valid operation.
- The adversary intercepts the communication and rearranges these message parts into z, y, x.
- The server processes these reordered messages because it thinks they are in the correct sequence.
- However, due to this altered order the server produces an erroneous result, leading to a denial of service for the user.

Modification:

- User sends a confidential message to the server “Share Project X’s details only with manager Tony”.
- Adversary intercepts the message and modifies the message to become “Share Project X’s details to everyone in the organization.

- The server receives the message and does not know it has been modified so it shares project X's sensitive information to tons of unauthorized members of the organization, breaching confidentiality.

How These Threats Could Manifest in Current Implementation:
All of these three sub-threats could manifest into significant attacks on our system, but for an illustrative example consider a **message reordering attack** on the Resource Server while it is undergoing an initial handshake with a new client.

- On step 2 of the handshake process, the client sends the Resource Server its newly generated {shared session key: k1} (encrypted with the Resource Server's public RSA key along with its {auth token: a1} (encrypted with k1)
- An adversary intercepts this request and reorders its body so now in the json shared session key actually corresponds to the authtoken, and the auth token corresponds to the share session key.
- The Resource Server receives the reordered message unassumingly and attempts to parse the authtoken's claims to ensure it was signed by the Authentication Server; however since it is actually now the 192 bit aes key an exception occurs, and the user's request could not be completed.
- The client can not advance in the handshake process with the Resource Server, leading to a Denial of Service.

Mechanism (*Figure 4)*
To protect against this threat group, we will add two main components to our existing mechanisms, as all encrypted client-server requests and responses will now be sent with an encrypted current Timestamp and HMAC. This means for all communication after the client has properly authenticated a Resource Server or Authentication Server (as the handshake protocols already utilize an encrypted timestamp), each original message will be concatenated with an encrypted current Timestamp generated by the sender. Then a base64 encoded HMAC will be appended to the end of the message by the sender, generated from hashing the encrypted message with the shared session key between both parties.

This will allow the receiving party to perform the same hashing technique originally done by sender (using the hashing function specified in Technical Choices), hashing the encrypted message they received and the shared session secret generated by the client during the handshake that is used for encrypting any request body, and then comparing the HMAC they computed to the HMAC code appended to the encrypted message. If the codes do not match, the receiving party can determine that this message was tampered with and its integrity has been compromised, detecting a potential attack (modification or reordering) and not moving forward with decryption to avoid potential dangers imposed by the attacker.

However, if the HMAC's are equivalent the receiving party can verify the integrity of the message, and begin decryption. Once the receiving party decrypts the timestamp in the message, it will then compare this timestamp to the current time. If the sent timestamp falls outside the time epoch specified by the server (specified in technical choices), the server will detect a replay attack and not move forward with processes the communication to avoid potential dangers posed by the attacker. However, if the timestamp falls in the specified time epoch, the receiving party can determine that no replay, modification, or reordering attack has occurred and move forward with processing the communication.

One additional mechanism that is in place to protect against out-of-order handshake requests is the resource server keeping track of what step of the handshake a client is on. If the client tries to make a request to an HTTP route that corresponds with the step of the handshake that the client is not on, the resource server will reject the request.

Proof of Security

To prove how this mechanism detects and protects against modification or reordering:
Utilizing the second preimage resistance and collision resistance properties of our specified hash function, it is computationally infeasible for an attacker to derive the same Hash output code from a different input. Under this assumption, let us also state that an attacker will never possess a shared session key from a properly authenticated Client-Server (resource or auth) session, as it is stored securely and is never transmitted unencrypted, or transmitted at all post-handshake. Thus, an attacker will not be able to obtain the input used by the sending party to create the HMAC output, that being the HMAC of the message and shared key.

Now the attacker may tamper with the original encrypted message itself, the HMAC code, or the message and the HMAC code. If the attacker has tampered with the original encrypted message, when the receiving party derives their comparison HMAC code from the message and shared key, it will not match the HMAC code in the message they received as these two instances of the hash function had different inputs, thus they will have different outputs (as aforementioned). The attacker may also elect to tamper with the HMAC code itself in an attempt for it equal the output of the HMAC of the message and shared key, but because the attacker does not possess the shared key it is computationally infeasible for them to create a new HMAC code that is the output of the HMAC of the message and shared key. This is once again because our hash function is collision resistant, making it injective for all computational purposes. Therefore, even in this case the receiving party's' comparison HMAC code it generates will not match the tampered HMAC code, leading it to detect a potential modification or reordering attack and halt normal processing.

To prove how this mechanism detects replay attacks
By ensuring that the sending party generates, encrypts, and concatenates a timestamp to the end of each message, we allow the receiving party to have a record of when this message was originally sent. Obviously, the receiving party would utilize the appended HMAC code to check against reordering and modification first, but if that check is passed the receiving party effectively verifies the integrity of the entire encrypted message, which includes the message's timestamp. At this point the receiving party can be sure it has an accurate record of when this message was sent, so now it can decrypt and check that timestamp against the current time. The crux of all replay attacks rests on the attacker sending a past communication in the present, but now the receiving party can easily check to see if this message was truly intended to be sent in the current moment, or if it was from the past, indicating a replay attack. The security of this mechanism rests of course on choosing a small enough time epoch to prevent very quick replay attacks.

Proof of Correctness:
Let us generalize and combine the sub-mechanisms we introduced for this threat together to draw conclusions about correctness. If a message is sent by a sending party, remains untampered in transit by any attackers, and is received in the present time frame as it was intended, the HMAC code comparison check and timestamp check performed by the receiving party will both pass. Thus, the receiving party can adequately confirm that no replay, modification, or reordering attacks occurred in this communication and proceed with normal processing of the message.

T6 Private Resource Leakage

Introduction to Threat:

In this threat, if exploited, a resource server could leak information to a principal that should not have access to it. Additionally, a resource server could leak information to a principal that once had access to the information but should no longer have access to it. As a result of this, sensitive information could be leaked to those who take advantage of the threat. This threat is problematic and needs to be addressed for several reasons.

1. Privacy Violation: Leakage of private resources can lead to a breach of user privacy. Users who use the system are expecting that their data is securely stored and only accessible to those with proper authorization.
2. Data Integrity Concerns: If an unauthorized user accesses data that they're not supposed to, not only could they read it, but they could also maliciously modify or delete information.
3. Reputation Damage: A security breach of confidential data could damage the reputation of the system and draw users away from the system.

Current Lack of Implementation:

In our current implementation, all files and data are unencrypted on the resource server. With the new threat model this would allow a malicious resource server to easily access to leak private resources with any unauthorized user they choose to.

Mechanism Description:

In order to protect against this threat all files will be encrypted using a symmetric key specific to the group. Files will be encrypted using the same method that communications are encrypted (192-bit AES with CTR mode and single use IVs). This key will be generated when the group is created and stored on the auth server. When a client authenticates with the auth server it will receive all the symmetric keys for all of the groups it is a member of. These will be stored only during the duration of the session. Before any authenticated request to either server the client obtains the most up-to-date version of the symmetric keys and a new auth token. A client will encrypt a file and its metadata before transmitting it to the resource server and decrypt it upon receiving it from the resource server. The resource server will store the files, the encrypted metadata, and the iv used for encryption (*Figure 5*). If a group member leaves or is removed, a new symmetric key is generated for the group and distributed to all users as they make another request or start new sessions. There are then two cases for re-encrypting the files for the group:

1. If any group member is removed from the group by the group leader, all files are immediately re-encrypted by the client that the group leader is logged into with the newly generated shared group key (*Figure 6)*. The group will be flagged to re-encrypt files on any resource servers that the group leader does not update.
2. If a user voluntarily leaves a group, the group will be flagged for re-encrypting their files. This would cause the client of the next user on a group to download all files and re-encrypt them with the new symmetric the next time they make any request to the auth or resource server. This could allow a user who gains access to an encrypted file to decrypt it before the files are re-encrypted, but any file that would access in this time frame was something they already had access to anyway. If they were to try to gain access to an encrypted file after departure, it would be a valid assumption that they would have already elected to download the file before leaving (*Figure 7*).

In either case, after the files are re-encrypted the client communicates the ID of the resource server they re-encrypted files for back to the auth server (these resource server ids will be discussed more in T7). If a user requests a token for a resource server that has not yet been updated, it will be sent back with the flag that suggests that the files for this group in this server must be re-encrypted. The auth server will have stored for each group the most recent key as well as a table of resource server ids paired with the key that was most recently used to update it. This table will only have entries for the ids of resource servers that have not yet been updated with the newest keys.

Proof of Efficacy:

This mechanism sufficiently addresses the threat as it encrypts files and file metadata with a symmetric key only known to the group members and Authentication Server. The encryption method is known to be a strong one with no practical attacks against it. Additionally, CTR mode without IV reuse is considered secure and efficient. Keys are rotated and files are re-encrypted upon a user's departure to ensure that the user who left the group will be unable to access any files. If a resource server were to leak any resources they would be encrypted and impossible to decrypt without knowledge of the shared key. Any clients that have an active session receive the updated shared group key at the start of their session and before any authenticated request to either server they make, therefore if they perform a request to obtain, upload, or update a file they have the updated key to perform the cryptographic operations.

Threat 7 Outline:

Threat Description:

If an illegitimate resource server steals a token belonging to a user, they can pass that token along to a legitimate resource server and impersonate the user. If a Resource Server is able to successfully impersonate a verified client using these tokens, they could potentially gain access to the files of all the groups the impersonated user is a member of. For this reason, it is necessary to ensure that tokens which have been stolen are not usable on any resource server except for on the server where the token was stolen from. Doing this makes stealing the token impractical because the attacker cannot use that token for a legitimate resource server to gain unauthorized access to resources.

Current lack of Implementation:

In our current application, this threat is not protected against. This can lead to exploitations if a fake resource server receives a token from a user believing they are accessing an authorized resource server. An attacker can take that token and send it to the real resource server, impersonating that user provided they have the victim's user ID.

Mechanism Description:

The mechanism chosen to protect against token theft is to generate an ID (*Figure 1*) specific to a given resource server at runtime and include the ID of the resource server the user is attempting to access as a JSON Web Token claim. Resource server IDs will be generated by using SHA-256 to hash the resource server public key and taking 16 characters (128 bits) starting from a randomized character. The user will ask a resource server for its ID in the initial handshake process between server and client. The server will then send a signed copy of its ID to the client along with the index of the key used to start the ID. The client can then verify the signature with the resource server's public key (which they have received out of band as described in the previous write up). Once the signature is verified, the client can hash the public

key and use the index to get the 16 characters to ensure that the Resource Server's serverID is what it should be. Upon receiving the ID from the resource server, the client will send a request to the authentication server asking it to generate a token for the resource server whose ID they have provided and send it back to the client.-At this point the user can move on with the handshake protocol and begin using the resource server (*Figures 2 and 3*).

Implementing our mechanism in this way means that a specific token will only allow the user access to a single resource server with a given token and will provide increased security against token theft. If the user wishes to make use of multiple resource servers, they can simply request another token from the authentication server for each resource server they intend to use.

Proof of Efficacy:

Our proposed mechanism fully protects against the threat of token theft because even if a token is created for a rogue resource server, an attacker will not be able to use that token for a legitimate resource server because the legitimate server will reject any request that comes with a token that does not have the correct resource server id, thus protecting the user from being impersonated.

If an attacker takes down a resource server and sets up an illegitimate server on that same port during a session between the server and client, the token will still not be successfully stolen. This is because server IDs are generated at runtime so when the old server comes back online, it will have generated a new ID and the token created by the auth server will no longer be verified for the server. An illegitimate resource server will be unable to fool the client into believing a false resource server ID because the ID will be signed with the resource server's private key and the protocol for the handshake between client and resource server in phase3 involves verifying that the resource server has the proper public key that was transferred out of band. Therefore, without the private key of the true resource server, an illegitimate server cannot pass a false server ID.

A rogue resource server will not be able to replay the handshake between the resource server and client to reuse the same id because the id is generated from the server's public key and signed with the private key. Since the key will only be accepted if the client can reproduce the key with the hash, successfully tricking the user into generating a token that would work on another server would require finding a keypair with a public key that would hash to the same value as the other server's key, which takes exponential time. Additionally, on the second step the client starts by sending a random nonce to the server along with its token, encrypted with the shared key. Even if a rogue resource server found a way to spoof the signature, it would not be able to decrypt the nonce without the private key.

Conclusion:

In this phase, there are two primary changes to our set of assumptions. First, we now need to protect our system from an active attacker rather than just a passive one. Second, the resource server is now so untrustworthy that it may leak our data and steal our tokens. These

changes lead to three new threats in our updated threat model. The first is that we now have far more possibility of replay, reorder, and modification in our communication to and from servers, which we have solved by using HMAC to verify that messages have not been changed and by using timestamps to prevent replay attacks. The second threat is that the resource server may leak our data, which we prevent by encrypting data with a key that is shared by members of a resource group. If group membership changes, we will generate a new key and the client will re-encrypt the data. Third, the resource server could steal tokens and use them on other resource servers, which we have chosen to prevent by making tokens resource server specific.

Appendix:

Figure 1: **Runtime Security Measures**

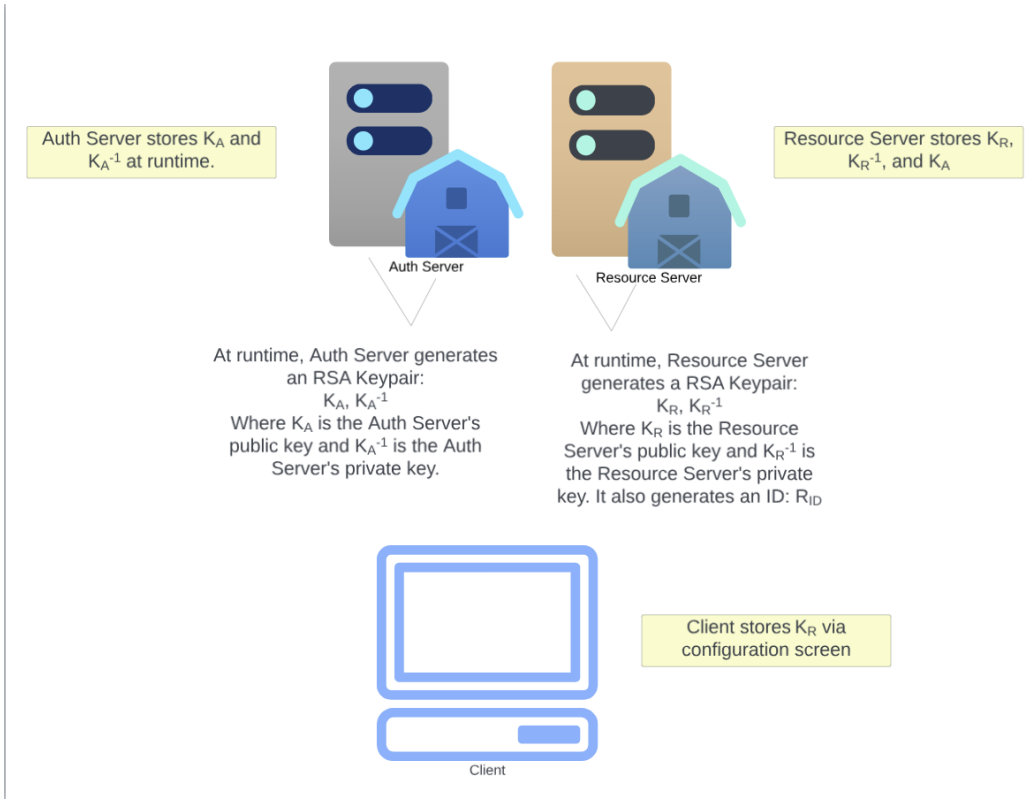

Figure 2: **Authentication Security measures (login or signup)**

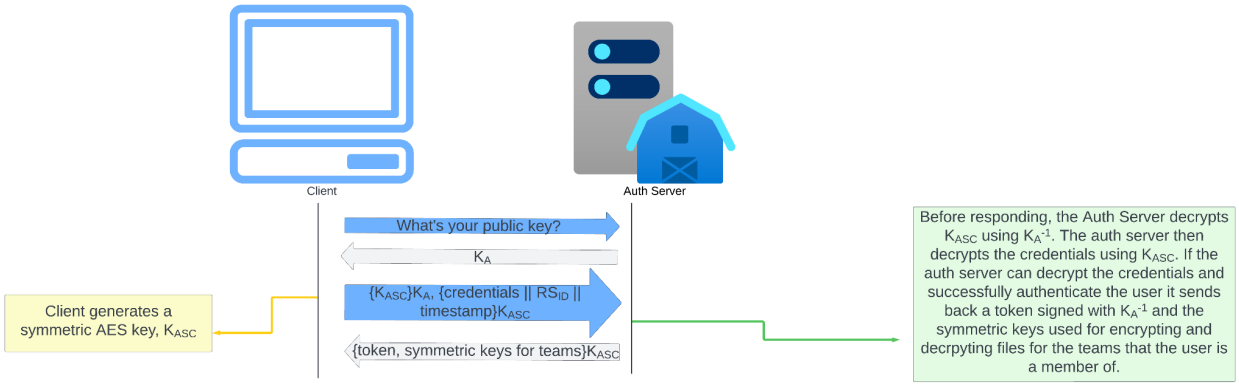

Figure 3: **Resource Server Request Security Measures**

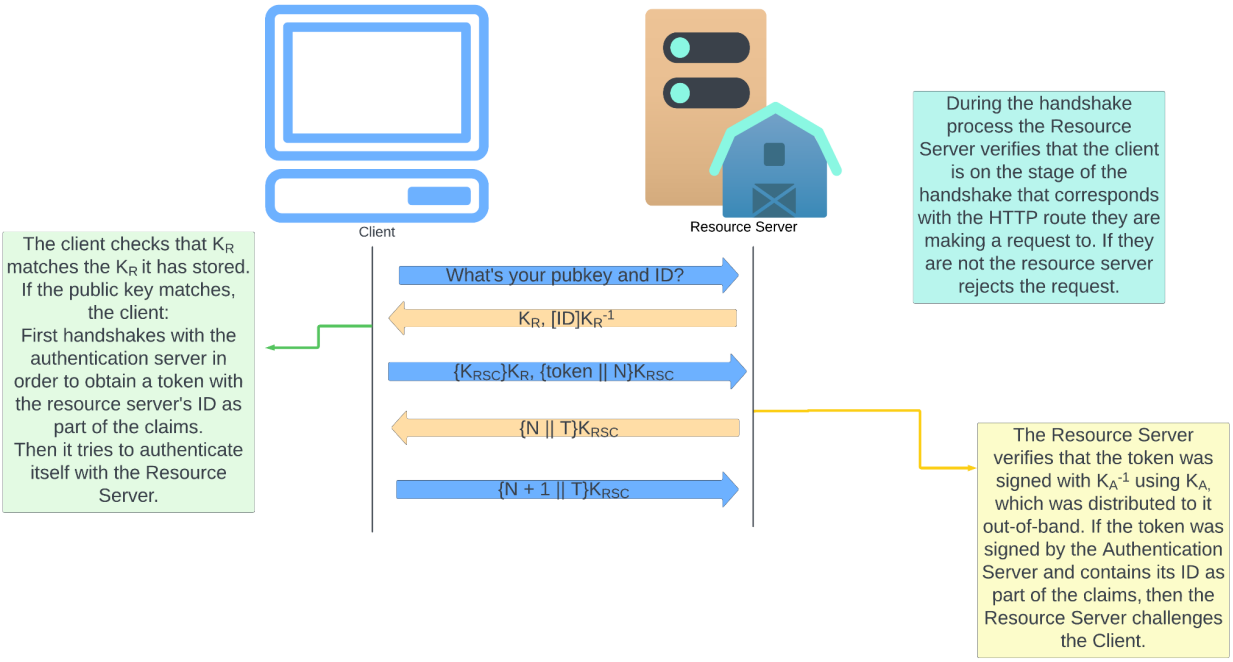

Figure 4: **Subsequent Communications Between the Client and Any Server**

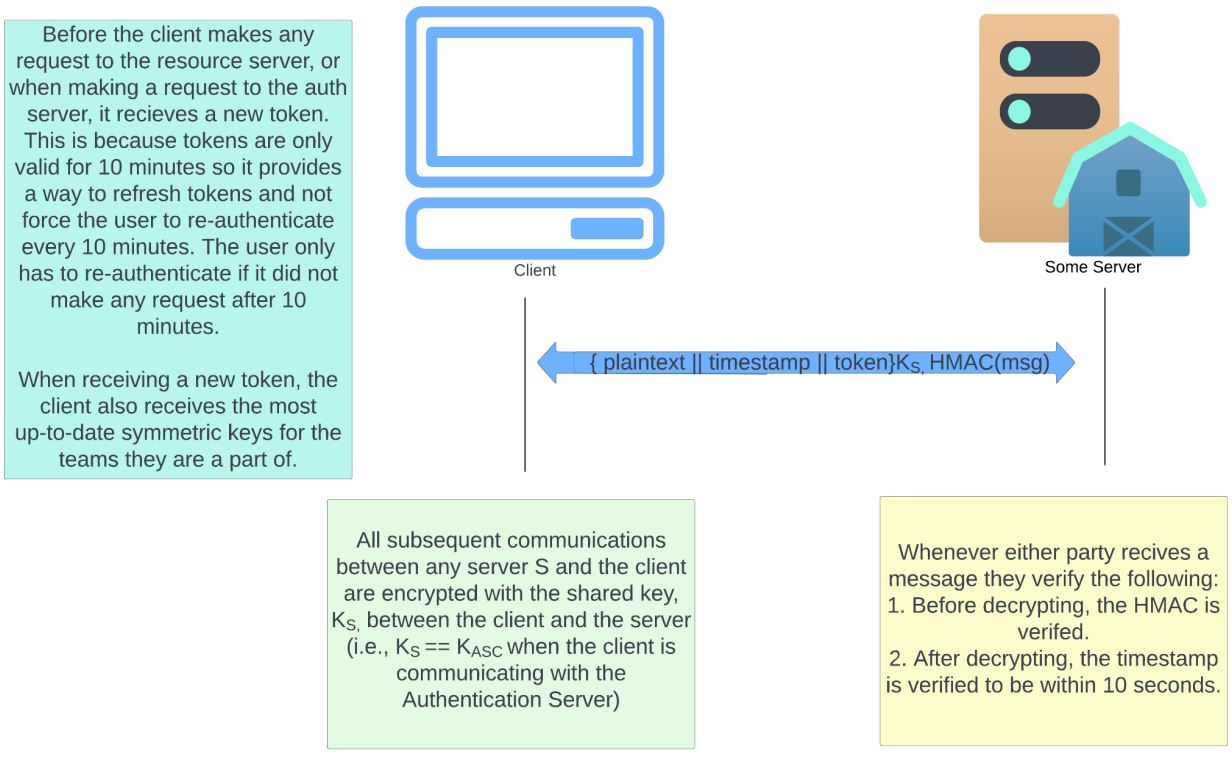

Figure 5: **File Encryption/Decryption Specific Measures**

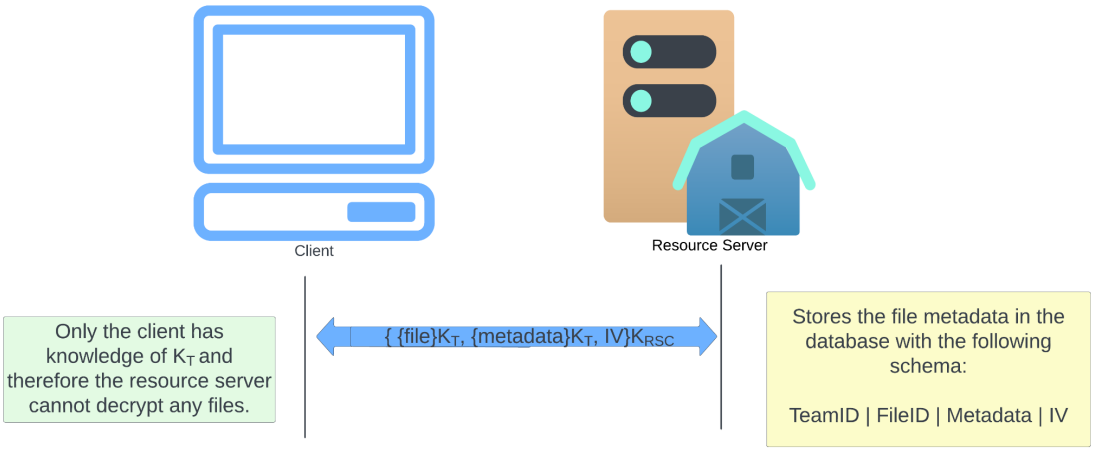

Figure 6: **Removing a User Case 1**

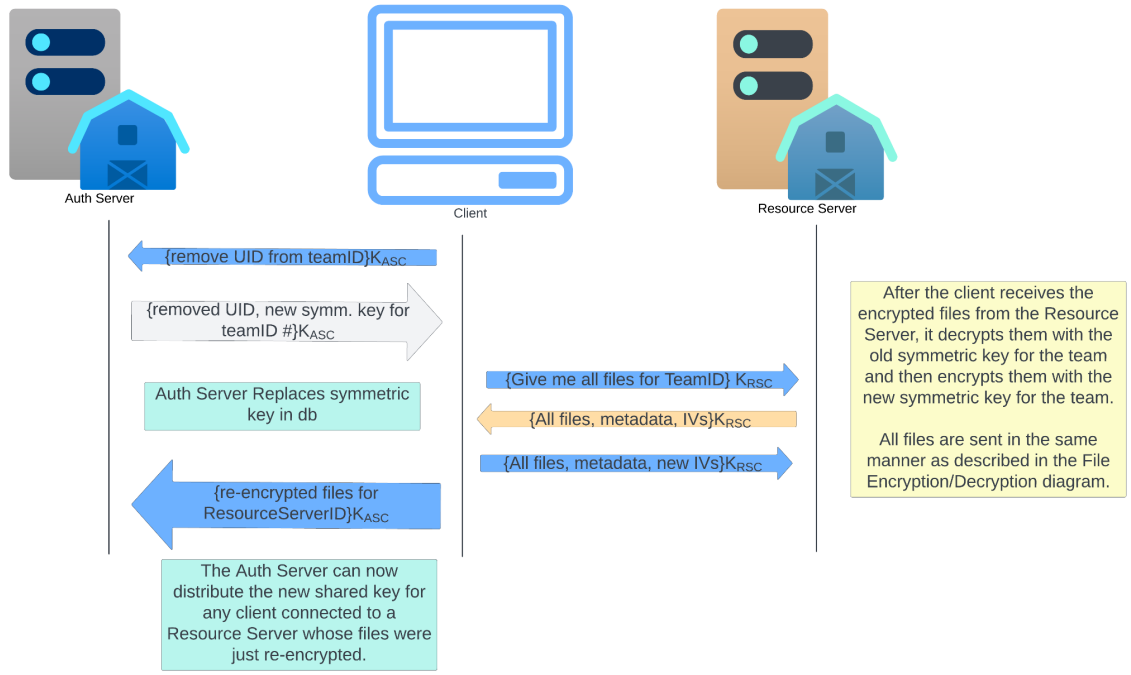

Figure 7: **Removing a User Case 2**