Phase 4 Mechanism Deliverable

The following content describes in detail the mechanisms for which message reorder, replay, or modification, private resource leakage, and token theft 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. We also assume that clients know the public keys of the desired resource servers ahead of time. This is reasonable because the context of our project assumes small groups that meet in person frequently.

Threat 5: Message Reorder, Replay, or Modification

Our system uses messages, defined by a message class, for all communication between servers. If an attacker were able to intercept a message between servers, we can reasonably assume they are unable to modify specific values within the message because it is encrypted before being sent, but they could still modify the encrypted contents or save the message to be replayed later.

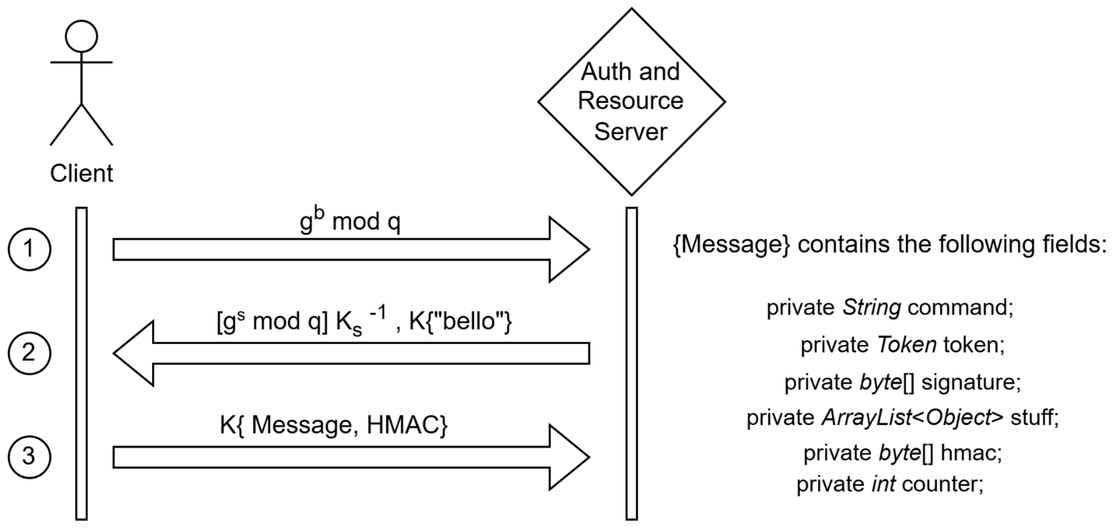
In order to prevent this from happening, we suggest two mechanisms: an incrementing message counter and an HMAC sent with every message.

```java
public class Message implements java.io.Serializable {
    // the command the Client is attempting to execute (e.g. “upload”)
    private String command;
    // a user token
    private Token token;
    // byte representation of the token signed with the AS public key
    private byte[] signature;
    // the content being exchanged (e.g. a file, if the command is “upload”)
    private ArrayList<Object> stuff;
    // the counter mechanism (detailed in T5)
    private int counter;
    /*
      byte representation of an HMAC generated by command, and counter
      and a symmetric key shared between the two communicating servers
    */
    private byte[] hmac;
}
```

Mechanism 1: Incrementing Message Counter
Within each message will be a counter that is incremented when the message is sent from one server to another. Both the Client and Server will keep track of this counter, which allows both to know what iteration the message is on. This prevents both reorder and replay attacks because if a message is not sent in the order it was supposed to be, due to either reordering or replaying at a later time, the server will immediately recognize this and can terminate the untrustworthy connection. Additionally, the HMAC will also contain the counter in order to ensure that even if the counter is tampered with, the message should still follow the correct sequence.

Mechanism 2: HMAC
Each message will also be sent with an HMAC generated from *String* command, *int* counter, and *int* session ID, using a symmetric key shared between the servers. We will serialize these fields into individual byte[], before concatenating them into one large byte[]. Then, we can generate an HMAC from this byte[] and the shared symmetric key. This symmetric key will be **different** from the session key, but likewise generated from the same shared secret.

This HMAC will allow recipients of messages to confirm that the message was unmodified in transit because if it was modified, the HMAC will recompute to a different value. Since both the authentication and the resource server use signed Diffie Hellman key exchange to create a shared secret, the process is identical for each server when generating a key for HMAC usage.


Client
Auth and Resource Server
1
g^b mod q
2
$[g^s \bmod q]\, K_s^{-1}$, K{"bello"}
3
K{ Message, HMAC}
{Message} contains the following fields:

```
private String command;
private Token token;
private byte[] signature;
private ArrayList<Object> stuff;
private byte[] hmac;
private int counter;
```

1. The client sends the authentication or resource server g^bmod q.
2. Since the server knows both g^bmod q and g^smod q, it sends back g^smod q with a signature and the arbitrary message "bello" encrypted with the new session key to show the client that it is a trustworthy source.
    a. Behind the scenes, the server generates two symmetric keys, one is a session key, and the other will be used exclusively for HMAC generation. It can generate these keys by applying subtle predetermined changes to the shared secret. For each additional secret

required, we will add 1 to the shared secret before generating the symmetric key with SHA256.

3. The client then verifies the integrity of the server's half of the shared secret and uses the new session key to verify the arbitrary message.
    a. Behind the scenes, the client also generates two symmetric keys, a session and key for HMAC generation, following the same process as the server
4. The client then sends messages with the command, the token, the signature, "stuff" (extraneous information depending on commands, such as login information, or files), the counter, and now the byte HMAC.

The client will generate separate HMAC keys for the authentication and the resource server. HMACs for both client and server will be generated for every message object. If the HMAC does not match, the session will be terminated.

Threat 6: Private Resource Leakage

Our current implementation does not take into account the fact that resource servers could be untrustworthy and leak files. Because we cannot guarantee this won't happen, we instead will proactively encrypt the files stored on the server so that any one resource server cannot directly provide the content of the files it stores.

Mechanism 1: File Encryption
For each group, all of the files that belong to said group will be symmetrically encrypted (SHA256) using a group-specific key that is stored on the AS and provided within its response to the Client. This means that a Client will be able to access and decrypt all files that are in the group they belong to using this key.

Method 2: Dynamic Group Key
Because users can be reassigned between groups, we must also plan for the event in which a user loses access to the files in a group. In order to enforce this revoked access policy, we will update the group key of a specific group when a user is moved out of that group in order to prevent them from continuing to access its contents.

However, this means that any files that were encrypted with the previous group key will no longer be able to be decrypted by the users that still belong to that group and should be able to decrypt it. Therefore, we will keep a full history of all group keys that existed for each group, which will be sent to the Client when requesting a token. This allows the user to decrypt all files that were encrypted within their group, regardless of which key was used.

In the case that the RS leaks files, a previous user of a group could still have access to the old keys if they saved them somehow, and therefore access whatever files they could before they had their group changed. However, due to the fact that they had the ability to download those files when they were a member of the group, we argue it is safe for them to still be able to see these files as long as they are not allowed to see any updates their old group has made (in the form of reuploading these files or uploading new files).

Mechanism 2.5: File Versioning
In order to determine which key should be used to decrypt any given file, we will include a version number with each file that specifies which group key in the history should be used for decryption. Each RS will contain a data structure that maps each file it houses to the file's version number. The version number of a file will be determined by the current length of the group keys array at its time of creation, which allows the file to be linked to the group key that was used to encrypt it. When the encrypted file is sent back to the Client who will decrypt it, the message will include the version number separately so that the Client knows which group key to use prior to decryption.

Mechanism 3: Token Expiration
In order to completely prevent replay attacks, we will also add a Timestamp value to the token itself that will guarantee the Token is fresh. If the Timestamp is older than one hour, the Token will be rejected and the user will be asked to re-verify their login information to acquire a new token. Normal users are highly unlikely to need to be continuously logged in to our system for more than an hour at a time, so this check should only inconvenience attackers.

```java
public class Token implements java.io.Serializable {

   private String group;
   private String username;
   private Timestamp timestamp;

   // Getters and setters...

   public String toString() {
       return username + ":" + group + ":" + timestamp.toString();
   }
```

Threat 7: Token Theft

As of right now, hashes of tokens are signed by the authentication server to prevent modification or forgery of tokens. However, a malicious resource server (after establishing a session with a client), could collect a token, and then pass it onto another client. This client could then go to a trustworthy resource server with this stolen token, establish its own session with the resource server, and then use the stolen token with its own commands to possibly manipulate data on the RS. **To prevent this, the token will now include a field called "id" which will represent the SHA256 hash of the resource server's public key (provided by the client.)**

Token Modification:
The client requests a token from the authentication server. Doing this, we will now require the client to provide the public key of the respective resource server they intend to connect to. After the client is

authenticated, the authentication server will include in the token an “id” which is the SHA256 hash of the resource server’s public key that the client had provided.

Verification Process:
Upon receiving a token, the resource server verifies the signature using the authentication server’s public key. Now, the resource server compares the hash of its public key with the id in the token. If they match, the token is accepted; otherwise, it is rejected.

Why it Works:
If an untrusted server steals and uses a token on another server without modifying it, the hashes will not match and they will get rejected.
If the token is modified to match another server’s public key hash, the signature verification will fail, and they will get rejected.

Thank you for reading our proposal!