Phase 4 Writeup

Crop Topographers: Ian Whitfield, Ojas Mishra, Jordan Brudenell, Lukas Finn, Riley Bode

Intro (Ian)

As in the previous phase, the following assumptions can be held for this phase unless otherwise stated:

1. Public or private keys refer to 4096 bit RSA keypairs.
2. RSA encryption will be done using OAEP padding with SHA256 as the chosen hash function.
3. RSA signing will be done using PSS padding with SHA256 as the chosen hash function. Furthermore, the signed material will be a SHA256 hash of the cited string, i.e. $[foo]_{A^{-1}}$ refers to the SHA256 hash of foo, signed with the private key A^{-1}.
4. Symmetric encryption will be done using AES, with 256 bit keys, 128 bit block size, in CBC mode with PKCS7 padding.
5. Every packet received *and* every decryption of a dictionary-like object should result in a well-formed JSON string, or the packet will be considered malformed and the connection closed.
6. Randomly generated nonces and keys will come from the OS's urandom implementation.

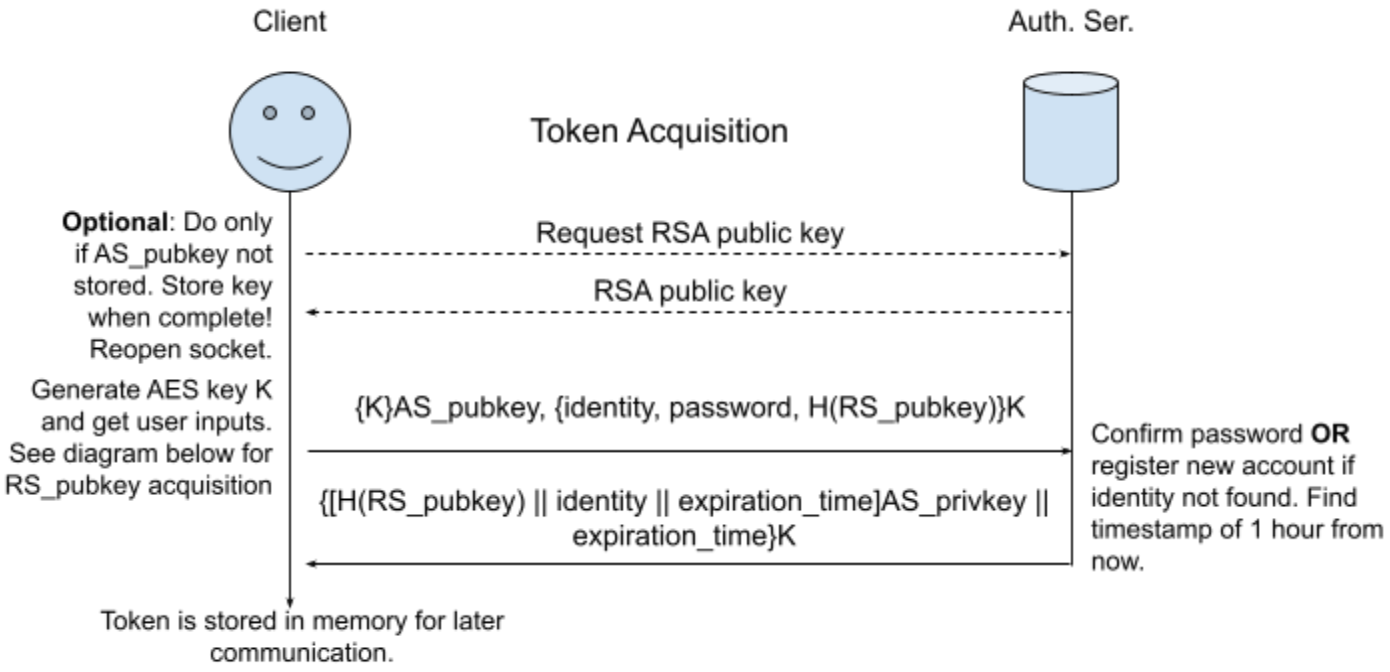
In Phase 4, the threat model has changed to enable more vectors of attack on the security of the leaderboard system. The first major change is that where there was once a passive attacker there is now an active one, meaning we have to take precautions against message reorder, replay, and modification attacks. The second thing to consider is a major reduction in the trust of resource servers. We are not only required to prevent resource servers from stealing user tokens and using them elsewhere, but also to assume that any information available to the resource server may be leaked to lower access level users.

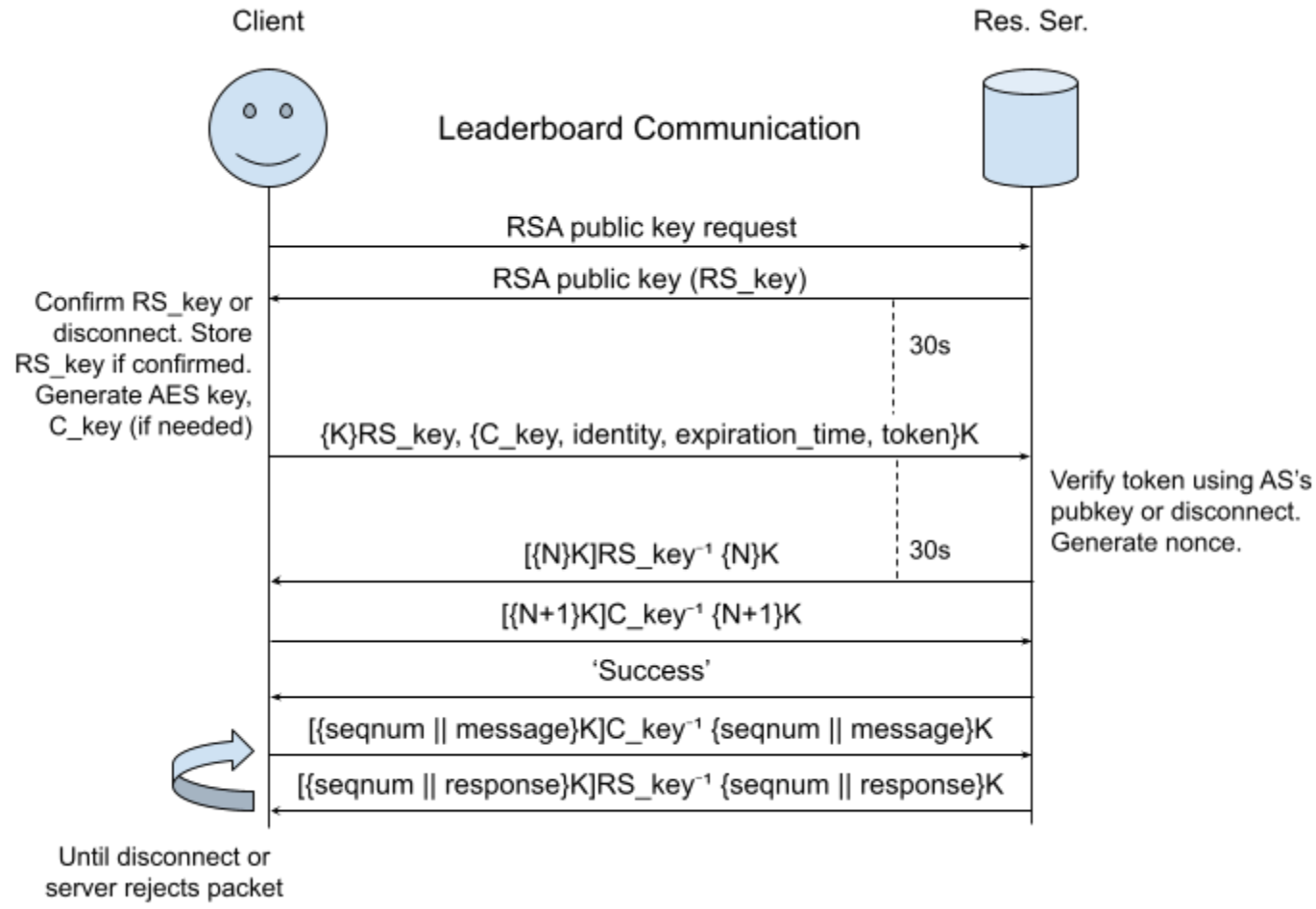
To address the active attacker's ability to intercept and handle messages nefariously, we can put an incrementing sequence number within the encrypted payload of the message. **Integrity is ensured by including a signature along with the encrypted payload, using the client's private key or the resource server's private key, depending on the direction of the message.** The nature of the authentication server protocol and the resource server handshake prevent them from being reordered or replayed, however, in the main body of message exchange with the resource server, a sequence number included within the encrypted packet *will* be required to prevent replay and reordering attacks – the communication follows a predictable request and response system so each party will always know what number to expect in the coming packet and will drop connection if invalid.

Theft of user tokens by a resource server can be addressed by including the hash of that resource server's public key **and an expiration timestamp** within the token's signed payload,

sending the timestamp in plaintext as well for verification. Should another server receive a token which isn't verified (with the authentication server's public key) to include the given identity and its own public key hash, **or is expired (using UTC for timestamp creation and verification)**, the token is rejected. To acquire such a token, clients will now have to specify the public key hash they would like to include when they exchange credentials with the authentication server, which will then sign the identity, the hash, **and the timestamp** as one token.

The most challenging attack to handle is that of the resource server leaking information to lower access level users. To simplify the mitigation of this threat, leaderboards no longer provide default access. Our mechanism starts with encrypting the score, comment contents, and proof contents for each leaderboard entry, which must be done with a symmetric key. Symmetric keys to decrypt the resources are stored on the server encrypted with RSA public keys associated with either a moderator group or an individual user. To "own" a symmetric key is for the server to store a copy of that key encrypted using a public key which you have the private pair to. **Each user has its own RSA key pair which is generated by the client application, sending the public key encrypted in the signin payload to the resource server.** Moderator group keypairs are generated by the admin, and are then treated as a symmetrically encrypted resource, thus requiring that an accessing user "own" the symmetric key as described before in order to act as part of the moderator group. Adding a user to a group gives them "ownership" over a key, and removing a user from a group means updating the key for that group with a fresh one, and incrementing its version number. Verified resources use the same symmetric key for each on a leaderboard, and unverified resources use a new symmetric key for each resource, one copy stored encrypted with the public key of the uploader and a second copy encrypted with the moderator group public key. **Each resource uses metadata to make it clear which key versions to use for decryption,** new or updated resources always use the latest key to encrypt.


Client
Auth. Ser.
Token Acquisition
Optional: Do only if AS_pubkey not stored. Store key when complete! Reopen socket.
Request RSA public key
RSA public key
Generate AES key K and get user inputs. See diagram below for RS_pubkey acquisition
{K}AS_pubkey, {identity, password, H(RS_pubkey)}K
Confirm password **OR** register new account if identity not found. Find timestamp of 1 hour from now.
{[H(RS_pubkey) || identity || expiration_time]AS_privkey || expiration_time}K
Token is stored in memory for later communication.


Client
Res. Ser.
Leaderboard Communication
RSA public key request
RSA public key (RS_key)
Confirm RS_key or disconnect. Store RS_key if confirmed. Generate AES key, C_key (if needed)
30s
{K}RS_key, {C_key, identity, expiration_time, token}K
Verify token using AS's pubkey or disconnect. Generate nonce.
[{N}K]RS_key⁻¹ {N}K
30s
[{N+1}K]C_key⁻¹ {N+1}K
'Success'
[{seqnum || message}K]C_key⁻¹ {seqnum || message}K
[{seqnum || response}K]RS_key⁻¹ {seqnum || response}K
Until disconnect or server rejects packet

T5: Message Reorder, Replay, or Modification (Lukas)

Threat Description

In this phase, we must consider the threats of an active attacker, rather than just a passive attacker. One of the resulting threats is that an attacker may try to intercept communications between the authentication server and another client. While these messages sent between the client and authentication server are protected from being easily read via encryption, there is still a concern that a malicious client could attempt to modify, replay, or reorder the messages sent between the client and server. If a malicious client is able to successfully replay or manipulate message order, it would mean that they can now resend a previous command sent by a higher-privileged user or deliver it out of order, possibly affecting its outcome. These are considerable threats to the system, as an attacker should not be able to gain any form of access or take any actions that they would otherwise not have as a normal user.

Security Mechanism

To counter this new threat, **all messages sent between the client and servers will contain a signature of the encrypted packet using the sender's private key. The recipient of the message will hash the received encrypted packet and will check if it matches the signature, closing the connection if modification is detected.** Additionally, all messages after the handshake between the client and resource server will be counted sequentially, with the message number passed as a field in the encrypted dict that is sent between the two parties. Starting at 0, each message sent will increment the message number field. For the entity sending the message, they will increment the number before sending and also store that number locally. For the entity on the receiving end, they will check that the number following their locally-stored number is equal to the received number. If this is not the case, the connection will be closed by whichever entity notices the inconsistency first.

Argument for Efficacy

This new mechanism sufficiently prevents the aforementioned threats in this phase. The focus of this mechanism was on preventing replay or reordering; packet modification is already covered by the fact that all messages are symmetrically encrypted during the body of communication. **By attaching the signature of the encrypted packet to the messages sent between the client and servers, we remove the possibility of message modification by easily detecting if any changes to the packet are made.** Similarly, attempting to modify them without knowing the AES key for the socket of the connection will result in producing garbage packets that do not decode to valid JSON and will lead the connection to be closed.

The implementation of the **message number field** is only concerned with the body of

messages sent between the client and the resource server after a connection has been made. Due to previous mechanisms, **all previous messages such as those necessary for the handshake protocol, are protected with a signature (preventing modification), have unambiguous ordering (preventing reordering) and use a challenge and response system where necessary (preventing replay), therefore not requiring a message number field within their encrypted packets.**

For all messages after the handshake, the message number field successfully prevents any reordering and replay threats. Since it is a field of the dict sent back and forth it is encrypted, meaning any attempted modification to the dict will be detected **by attempting to verify the signature and** by the JSON parser, and the connection will be terminated. If an attacker were to try to replay a previous message, both the server and client would be able to detect the re-used message because of an outdated message number field, and the connection would also be stopped. Similarly, if messages by one entity were intercepted and sent reordered to the other entity, the inconsistent number would be detected, closing the connection.

T6: Private Resource Leakage (Jordan)

Threat Description

Resource servers are untrustworthy to the point that any resources seen by the server must be assumed to be potentially leaked. Therefore, the resource servers must never possess any resources which are intended to be private between members of a group. Furthermore, when group membership changes, the secrecy of those resources must be maintained to exclude a user who is removed from the group.

Security Mechanism

The authentication server will now provide a public/private RSA keypair to users for use on resource servers.

Read groups will have a symmetric key associated with them, shared with users by encrypting the key with the user's public key. Data intended to be readable by a read group will be encrypted with this key.

Moderator groups will have an RSA keypair associated with them, shared with users by encrypting the private key with a symmetric key, then encrypting that symmetric key with the user's public key. When a user submits an entry, it will be encrypted with a newly generated symmetric key, and that key will be encrypted with the moderator public key. The user will also encrypt that symmetric key with their own public key and store both encrypted keys alongside the data.

The encrypted keys will be stored on the resource server along with a version number. When a user is removed from a group, a new group key will be provided to all the members of that group by adding the key to a sequence of used keys. Then, when entries are updated or added, they will use the new key, but old entries will remain encrypted with the old key. The version of the key used will be stored alongside the encrypted data. **The server can handle**

selection of a key by checking for the versioning metadata and searching for a key entry for that resource, user, and version, and responding with an Authentication Failure error code if not found, otherwise including the found key in the response.

Argument for Efficacy

Because clients will be encrypting all data that they send to the resource server, it is not possible for the resource server to leak any plaintext data. When a group is created, the user doing the creation creates the initial keys to be associated with that group, and the server never has access to those plaintext keys.

We accomplish this secrecy through a mix of public key and symmetric key encryption, using public keys for key distribution and symmetric keys for encryption of data. Each user has an individual RSA keypair that is used to encrypt the symmetric keys used for data encryption, as well as the symmetric keys used to encrypt other RSA private keys when it is necessary to share them (e.g. when a user is assigned moderator status).

When a user submits a new entry, they need to be able to encrypt that data such that only they and the moderator group can see it. This requires that the moderator group have an associated public key accessible by the user, which they can use to provide the freshly generated symmetric key used to encrypt the data.

Then, when a moderator views and verifies that entry, the associated data can be re-encrypted using a key known to all users that have permission to view that leaderboard.

Both the moderator level RSA keypairs and read level symmetric keys need to be updated when a user's privilege is revoked. The existing data is only lazily re-encrypted, when modified or new entries are added. Each user added to a group gains access to all past keys, ensuring that they can view all data currently present.

T7: Token Theft (Ojas and Riley)

Threat Description

Resource servers are considered to be very untrustworthy in this phase, so we must consider the possibility that a resource server steals a user's authentication token and will attempt to connect to other resource servers with the token given to them by a user. In order to ensure that this cannot happen, authentication tokens will need to be unique to each resource server.

Security Mechanism

We implement the binding between an authentication token and a resource server, via getting the authentication server to sign the hash of the resource server's public key. When a client is acquiring an authentication token, the client now appends the resource server's public key along with their username and password. The AS returns the signed hash,

[identity || SHA256(RS_pubkey) || **expiration_date**]AS_priv which acts as the authentication token for the RS.

If the RS sees that the RS_pubkey provided to them by the client in the token doesn't match their own public key, they immediately drop the connection.

Note that the client acquires RS's public key via out of band communication, and they confirm its integrity via a visual glance of the hash before connection starts.

Every token contains a timestamp that indicates the time of issuance, with the timestamp being sent in plaintext alongside the token. If a resource server encounters a token that is more than an hour old compared to the current timestamp in UTC, then said token is expired, and the resource server will reject the request with a Token Expired error code.

Argument for Efficacy

Since a resource server's public key and private key are randomly generated, it is reasonable to assume that each resource server has a unique public key, and thus a resource server can be uniquely identified via its public key.

The new auth token creates a binding between the server being connected to and the auth token itself. This binding prevents auth tokens to be used for any server other than the intended server, meaning that a stolen auth token is of no use to the malicious resource server.

This binding is unforgeable and immutable as the token is signed using the AS's private key, meaning that only the client (the party providing the RS public key) and the AS (the signer) can modify the binding, who in this case are both trusted.

This binding ensures that whenever a malicious RS acquires an auth token from the client, they cannot connect to another RS using the same token, as the other RS will have a different public key, and therefore require a different token.

Tokens being expired after 1 hour of issuance means that a third party who has acquired the token, can't use it indefinitely.

In conclusion, a token issued by the authentication server ensures that a particular identity can connect to the specific resource server associated with the token for the next hour, and no longer.

Conclusion (Ojas)

The new additions to the threat model were the active man in the middle as well as a potentially malicious resource server. We address the man in the middle in T5. Since all potentially vulnerable messages sent over the channel now include a monotonic counter, a man in the middle can't obtain information about a valid message that can be sent, even if the MITM knows all previous encrypted messages.

The capabilities of a malicious resource server that would affect us are as follows.

1. The resource server has access to all messages sent to it. This threat is addressed in T6.

2. The resource server has access to the auth token sent to it by the client. This threat is addressed in T7.

Since we cover all capabilities of an active man in the middle, and a malicious resource server, we have secured ourselves with respect to the extended threat model.