Phase 3 Writeup

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

Intro

We begin with a clarification of certain items. It shall be assumed that unless otherwise stated:

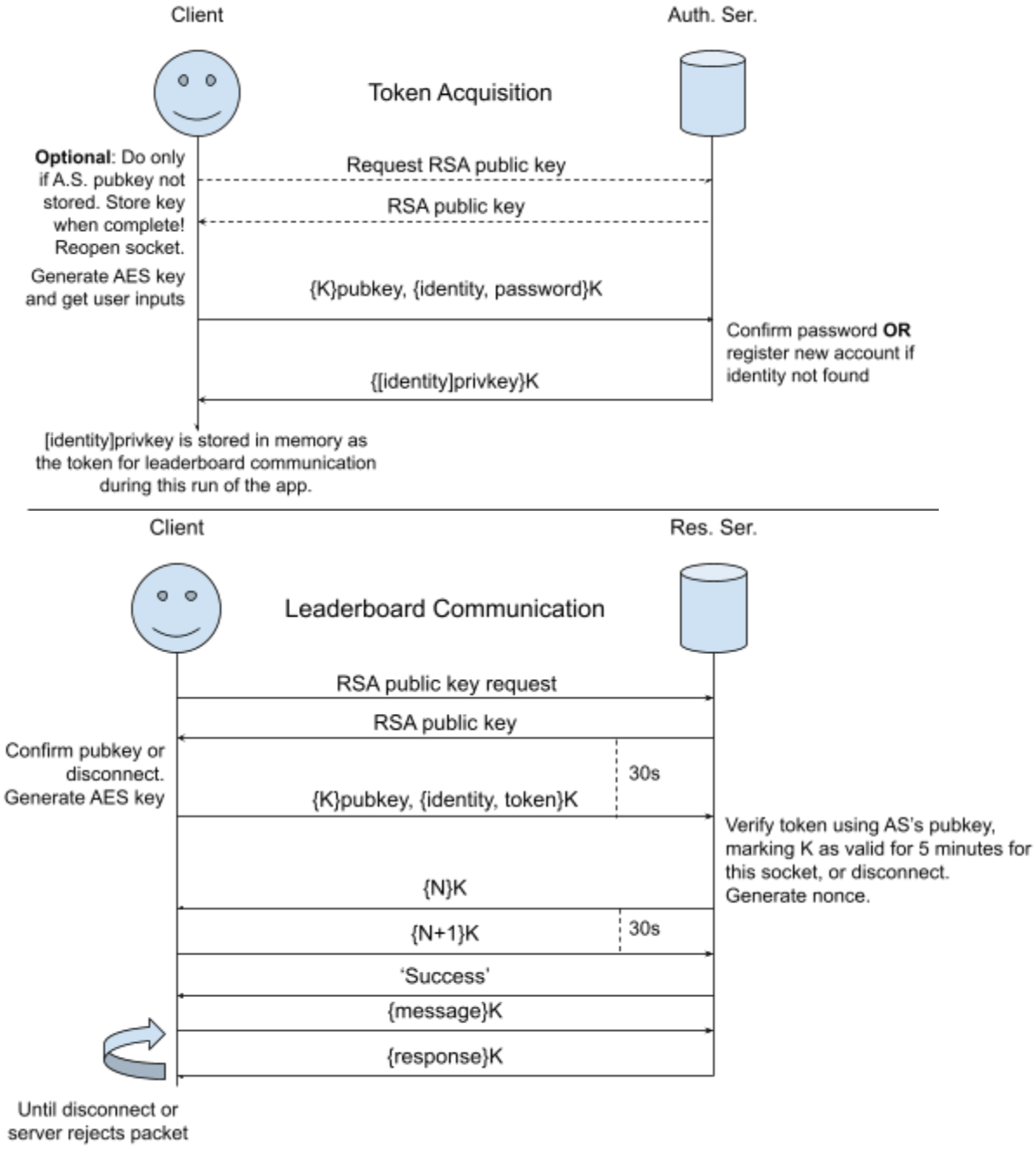
1. Public or Private keys refer to 4096 bit RSA keypairs.
2. RSA encryption will be done using OAEP padding with SHA256 as the chosen hash function.
3. RSA signing will be done using PSS padding with SHA256 as the chosen hash function. Furthermore, the signed material will be a SHA256 hash of the cited 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.

Our system is characterized by the authentication server knowing nothing about what permissions a user has for a particular resource server. The authentication server's only purpose is to verify that a user is who they say they are and to provide the user an identifying token which can be used to log in to a resource server.

In order to respect the authentication server's authority, it is necessary that a resource server has a copy of the authentication server's public key which they have confidence in. For our purposes, we will assume that the authentication server's public key is made available out-of-band and that a user who wants to create a resource server will be able to access the authentication server's public key through some kind of secure channel, since they want to utilize its services to allow users to log in.

The authentication server's communication protocol depends on the client knowing its public key, so a client may first need to request and receive the public key from the authentication server in plaintext, otherwise acquiring the key out-of-band. The client is then to send an AES symmetric key encrypted with the server's public key, as well as a payload encrypted using the aforementioned AES key which contains the information necessary to verify their identity to the authentication server. Once verified, the auth server will use that same symmetric key to encrypt a login token for that user which is simply an RSA signature of the SHA256 hash of that user's identity.

Once a user has their login token, they may attempt to connect to a resource server. That connection begins with the client requesting and receiving the resource server's public key. The client will then verify that the public key matches the copy they have on file (if one is present). If the client has no public key on file, the user will be presented with a SHA256 hash of the public key and asked to compare it against the hash provided by the server owner out-of-band, saving the key to disk if confirmed. After verification of the key, the client will send a new AES key, encrypted with the resource server's public key, concatenated with the client's identity and token, which have been encrypted with the AES key. Using the authentication server's public key, the server will verify that the token is a signature of the provided identity. If the token is verified, the server will respond with a 256 bit randomly generated nonce N, encrypted with the provided AES key, and the client will be expected to respond with N+1, again encrypted. After the server receives that correct response it will respond with a success message and expect further issued requests on this socket to be encrypted with the AES key, now referred to as the session key.



T1: Unauthorized Token Issuance

Threat Description

This threat is new to this phase primarily because our trust model has been changed. In the first phase, clients were expected not to lie about their identities when authenticating, which enabled us to send them back a plaintext token that mirrored their identity. In phase 3, clients are no longer trusted, and because we have no way of authenticating users to verify that they are who they say they are, any client can choose to act as any user on the server.

This can cause a few issues, the first of which being that it violates the security properties of the system which require users to truly be who they say they are. More materially, a user that is not allowed to access a resource will be able to log into another user account which *is*. Our system makes it somewhat difficult to view who has what permission, but in the scenario where logging into another user's account takes very little effort on the part of the attacker, a brute force process of logging into each of the very finite number of users listed on the server could be catastrophic.

The most critical threat to the system caused by untrusted users is the potential for a user to log in as the singular admin account, which is the same across the entire resource server. Admins have all permissions across all leaderboards, also leading to concerns about separation of privilege/single point of failure, regardless of authentication implementation.

Security Mechanism

Our mechanism for addressing this threat is to implement a secure password authentication scheme between the client and the authentication server. The end goal of the scheme is to provide a token back to the client which can be used to prove its identity when provided to resource servers in later exchanges during that run of the application.

The first step to obtaining a token from the authentication server is to have a copy of its RSA public key. A client that has not yet stored a public key for the current authentication server will make a cleartext request to the authentication server for the public key, and the server will respond with the public key immediately upon receipt. This will be stored on disk and associated with the authentication server's information. The user does not have to know anything about this process happening, it will occur before any further communication occurs with the authentication server, and usually only once. It goes without saying that the authentication server should generate an RSA keypair when it first creates its local database, and keep it for as long as it is uncompromised. Users are required to delete their own copies when a public key changes. The socket is **closed after each request**, so the user must reopen it.

The client then generates a random AES key and asks the user to provide an identity and password to log into or register with the authentication server. The identity and password are encrypted using the AES key generated before. This ciphertext is concatenated with the AES key itself, encrypted using the RSA public key acquired in the first step. The complete message is sent to the server.

Upon receiving the message, the authentication server decrypts the AES key using its RSA private key and uses the AES key to decrypt the identity and password pair. If the identity is not registered as a user in a local SQLite database, **its entry is created with the given password**. This means that your first **login attempt** with a given identity will be what sets your password permanently. A login attempt for a nonexistent user will always succeed in returning a token. A login attempt for a preexisting user will require that the authentication server check the provided password against the password stored in the entry associated with that identity. In either case, **unless there is an existing identity and the password comparison fails**, the authentication server will sign the identity using its RSA private key and further encrypt the signature using the AES key from before. This is sent to the client, who will **decrypt the signature which acts as a token for later exchanges with the resource server**. The token doesn't expire in this phase, so hypothetically you would never need to connect to the authentication server again if you store the token to disk, though our implementation does not do this, and instead keeps it in memory for the proceeding connection to a resource server. The authentication server will immediately close the connection after delivering the encrypted token.

Argument for Efficacy

The first step of this protocol is a cleartext public key exchange. No information can be gained through passive listening that wouldn't be available otherwise, and given our current threat model we have no reason to believe the public key we receive has been forged or comes from an attacker rather than the authentication server.

The second step requires that we secure the communication channel between the client and authentication server so that the secret password and token can be exchanged. We chose to use a symmetric encryption protocol to encrypt the main body of communication because it allowed us to do fewer checks on the length of packets and is more efficient on its second use than the equivalent RSA-only encryption would be. Specifically we chose AES-256 as it is the popular standard for symmetric encryption and we have no concerns about its performance. To transmit the symmetric key, it is encrypted using the authentication server's RSA public key, meaning only the holder of the paired private key can decrypt the contents. The identity and password are encrypted with AES using the symmetric key. These measures mean a passive listener would need the RSA private key or the AES key to gain any knowledge of secrets. **Replaying the login/registration step as an attacker would grant you the same response from the authentication server, but if you weren't the one to generate the AES key, you would have no way to decrypt the signed identity to send it as a token when connecting to a resource server. Nothing else is encrypted using this instance of the AES key.**

In the final step, we have no reason to protect on-disk data, so the identity and password are stored in cleartext on the authentication server's database. The signed identity acts as secret because it can only be produced by the authentication server, and thus can only be acquired (by clients) by having a valid password for the signed identity. If a resource server is given this secret, it can be verified using the authentication server's public key and an identity to compare it against. This confirms that the secret holder has the password for the provided identity, and therefore, for our purposes, has the right to act as that identity.

T2: Token Modification/Forgery

Threat Description

With clients no longer being trusted, we have to now assume that users who are incentivized to increase their privilege will attempt to do so. A user should only be able to execute commands that are allowed by their privilege level on a given resource server or leaderboard, depending on the specificity of the request. This requires that they send, along with their request, some indication of their identity which enables the server to check their permissions in the database.

The threat that this section addresses is that a dishonest client could send a forged or modified identity to the server, allowing them to use the permissions associated with a more privileged user to authorize their request. Therefore, there is a need to protect the integrity of the identity message, somehow ensuring that the client is authenticating at the resource server with the same token they would receive at the authentication server.

Security Mechanism

Our implementation verifies integrity using RSA signatures. After a user has passed the password check, the authentication server sends a signature (using its private key) of the hash of the identity string given by the client. The authentication server's public key is obtained by the resource server via out of band communication, so that the resource server can later verify that a token given to it by a client was originally generated at the authentication server.

Argument for Efficacy

Any malicious user who wants to falsify their identity for any reason will also have to falsify the corresponding authentication server signature in order to log in with that identity. Forging the signature requires knowledge of the authentication server's private key, however because this is not considered for this phase, the only alternative would be a computationally intractable brute force.

A user could also attempt to create an identity which hashes to the same as the identity they wish to log in as, eventually producing the same signature, however our hash function is believed to be preimage resistant such that an attack of this nature would also require a computationally intractable brute force.

T3: Unauthorized Resource Servers

Threat Description

With the update to the threat model, now only authorized resource servers are completely trusted. This does not necessarily mean that resource servers are listed somewhere

in a centralized database, but rather states that the user must verify that the server they are connected to is the intended one. Without verification, it would be possible for attackers to pose as the original resource server without the client knowing, intercepting data or otherwise lying to clients given their spoofed identity.

Security Mechanism

To counter this threat, the first step is in the resource server. At the server's creation it will create an RSA keypair. The server owner is then expected to distribute the hash of that key out-of-band for verification purposes. When the user attempts to connect to a resource server, the client will first request and receive the server's public key in plaintext.

To ensure the resource server is correct, the user will compare the received public key to the last stored public key for this particular (intended) resource server. If this is the first time connecting to that server, the user will be asked to verify the received public key by comparing the hash of the key with the hash provided by the server owner. If the hashes match, the client will generate a new AES key. This key will then be encrypted using the previously mentioned public key and sent to the server, establishing a symmetric key for the proceeding session between the client and server.

Argument for Efficacy

This solution provides security in assuring that a user is connecting to the resource server that they intend to. During each connection attempt, the RSA public key provided by the resource server is checked against the last-stored value of that server's public key, ensuring that we are connecting to the same server as the last time we tried to connect to that IP. Upon first connection, if no key is stored for that resource server, the key is verified by comparing the hash of the key to the hash provided by the owner of the resource server, giving the client confidence that this is the correct public key to store. When the client sends the symmetric key encrypted using the server's public key, only the verified resource server can decrypt the message, making the symmetric key a secret shared by only the client and the server for this session. An attacker who provided the correct public key would not be able to decrypt the symmetric key, and an attacker who provided their own public key would not match the key stored on disk.

The success of this security implementation is dependent on the owner of the server and the user attempting to connect to it. For this implementation to be secure, it is vital that the first connection made can be verified as the correct server, as it relies on a previously-stored key to ensure that the server isn't lying about its identity. Since there is no last-stored key on first connection, the owner must give the hashed key to the client user so they can properly compare it to the first transmitted public key and its hash. If the owner decides to not share the hash, then the user has no choice but to assume that the public key sent by the resource server is to be trusted. Similarly, if the owner sends the hashed key to the user and the user decides to not compare hashes, they are again assuming that the resource server is trusted and is sending the correct public key. Either way, a lucky attacker who impersonates the server during the first

connection would become the de-facto true identity of the server, furthermore blocking the real server from ever being connected to.

T4: Information Leakage via Passive Monitoring

Threat Description

This threat is present during all communications between the client and a server. The first way this threat can cause issues is with unencrypted information leaking which we intended to keep secret. Should a sensitive message be transmitted unencrypted, or encrypted weakly, a passive listener would violate the confidentiality of the communicating parties. Even if communication were to be encrypted, semantically insecure communications could also allow a passive listener to gain some degree of information, and the complete collection of messages also means a replay of these messages would be possible. Thus, replay attacks also need to be considered as part of this threat model. A replay attack could cause any number of issues, ranging from confusing a server with previously sent but invalid data, all the way up to repeating commands executed on the server, such as repeating a post made by a user or deleting all of their posts one by one.

Security Mechanism

In communications with the authentication server, the protocol described in Threat 1's security mechanism takes care of any passive monitoring threats in the authentication process. Every time a user tries to authenticate themselves, an AES key is generated by the user and sent encrypted using the server's public key, and the token sent by the authentication server to the user is encrypted using this key to prevent anyone lacking the random AES key or the server's private key from ever having a chance to decrypt and see the token.

In communications with the resource server, an AES key is shared by the client in a similar manner, however the client's identity is instead verified by passing their token rather than a password. The resource server then verifies that the token is a valid signature using the authentication server's public key with the identity provided. The resource server then generates a nonce and sends it to the client encrypted with the symmetric key as a challenge. The user then sends back the nonce plus one encrypted with the symmetric key back to the server. The challenge process must be completed with a 30 second timeout in place, in other words, a client who doesn't respond for longer than 30 seconds will be considered inauthentic and disconnected. From there, the resource server and the client use the symmetric key to encrypt **all** communications until the pipe is broken. After 5 minutes without a request, the resource server will send a session expiry message and break the pipe. The purpose of this timeout is primarily to prevent very long lasting idle sessions from weighing down the server, and for this phase it has no significant security impact. As a residual effect, the timeout could be considered useful in keeping people from leaving a session open forever and thus encrypting all messages with the same session key, which may at some point be compromised.

Argument for Efficacy

In communications with the authentication server, because a private AES key is generated by the user, the encrypted token sent by the authentication server to the user cannot be used by an attacker without the client's generated AES key. This also prevents any possible replay attacks. While resending the message that was sent from the client to the authentication server will result in a response, without the AES key, the token cannot be decrypted and therefore cannot be used when trying to connect to a resource server.

In communications with the resource server, the exchange ensures that nothing about the key and the token can be discerned without having access to the resource server's private key or the symmetric key. The nonce used as a challenge ensures that a replay attack is not possible without knowledge of the symmetric key, as the challenge used by the resource server has a 1 in 2^256 chance of using the same nonce, and an attacker cannot solve the challenge without access to the symmetric key if there is a different nonce used. The purpose of the 30 second timeout during authentication is that a 5 minute timeout, which is standard throughout the rest of the protocol, wouldn't make sense during parts of the communication which require no manual user input. 30 seconds should be plenty of time, and giving the least leeway possible without losing a large number of users seems like a good general practice.

Conclusion

Our design process began with a survey of the four threats and a generalized approach to each of those individually. Initially it was our thought that each threat would be solved in isolation but in a further brainstorming session we developed a protocol which attempted to address threats not presented in this phase. That protocol was stripped of anything extraneous until we arrived at our current design.

We often found ourselves unnecessarily trying to design around active listeners who would be able to capture and modify traffic, such as sequencing requests to prevent replay attacks, or trying to prevent a resource server from being able to impersonate a user on another resource server.

Our biggest hurdle was rationalizing a way for the resource servers to verify something as being from the authentication server without the resource servers ever talking directly to the authentication server, but in the end we reasoned that it makes sense for a resource server wishing to use a the authentication server should know about that server's public key somehow. We further decided that setting up and configuring the resource server would include this simple step.

A second hurdle was that our initial understanding of the unauthorized resource server threat was mistaken and allowed for a man-in-the-middle attack to easily take place against a resource server. After revisions, we believe we have made a more sensible protocol.

Ensuring that listeners cannot learn things which must be kept secret and cannot enact replay attacks necessarily overlaps with everything else, but other threats in this phase and our solutions to them are discrete.

Bonus Round

It's worth clarifying in writing how resources and users are grouped on our system. Each leaderboard is given a name, and each leaderboard has multiple permission levels specific to that leaderboard. In this way, you could look at each permission level within a leaderboard as a named group, with respect to the server as a whole. Users added to the Read permission group for a leaderboard have read access to all of the verified entries on the given leaderboard, entries being our primary resource. Write permission for that leaderboard gives a user the ability to add entries to the leaderboard, in other words, adding resources to the resource group. Moderator permissions enable a user to see and remove verified *and* unverified entries for a given leaderboard, as well as move unverified entries into the verified category. Having any one of these permissions for one leaderboard has nothing to do with your permissions on another leaderboard, which is why these permission groups can be considered named groups (each group is distinct from the next, delineated by the leaderboard name and the permission level).