Phase 5 Writeup

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

Intro (Lukas and Ian)

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

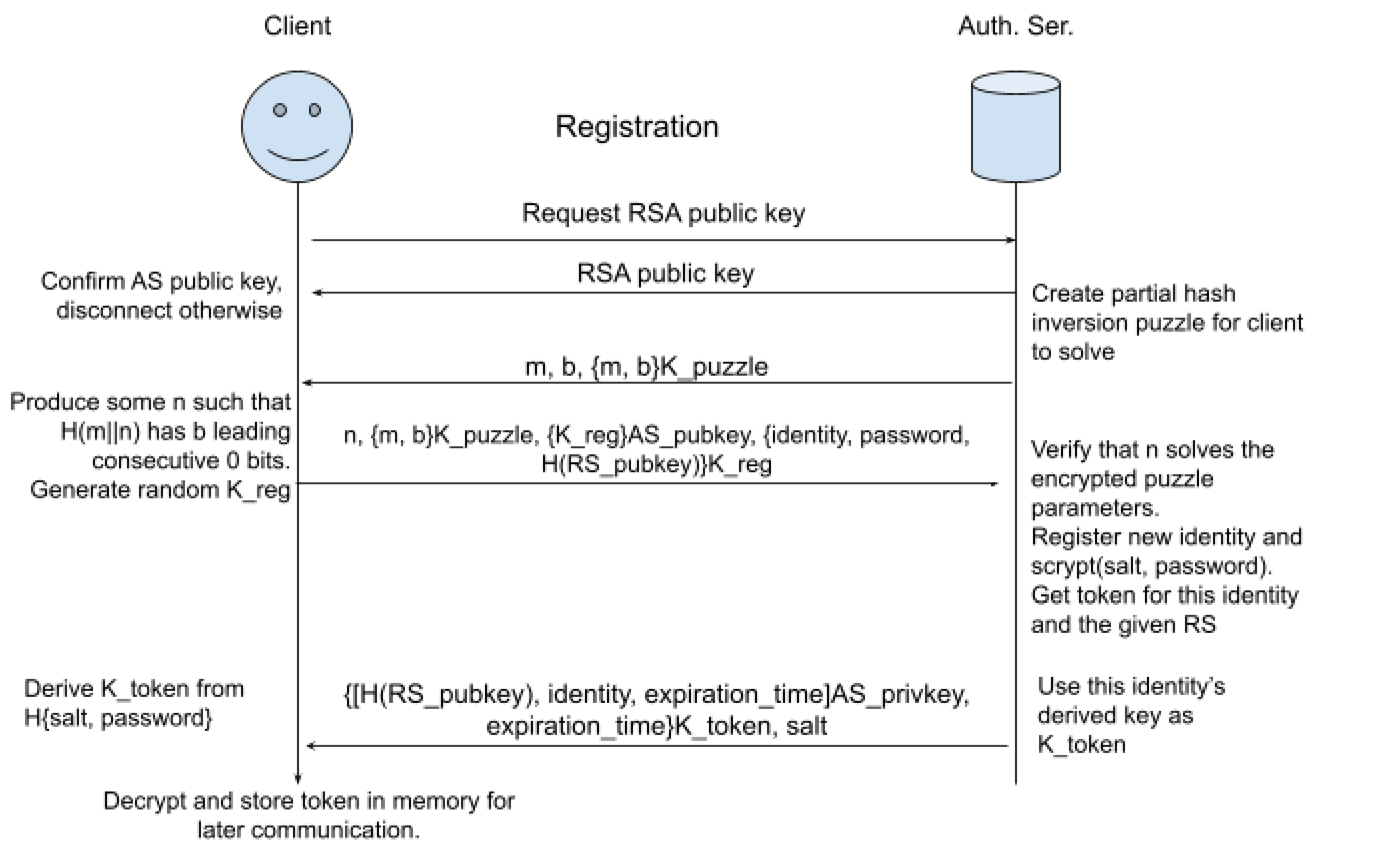
1. Public or private keys refer to 4096 bit RSA keypairs.
2. RSA encryption will be done using OAEP padding with SHA256 as the chosen hash function.
3. RSA signing will be done using PSS padding with SHA256 as the chosen hash function. Furthermore, the signed material will be a SHA256 hash of the cited string, i.e. $[\text{foo}]_{A^{-1}}$ 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.

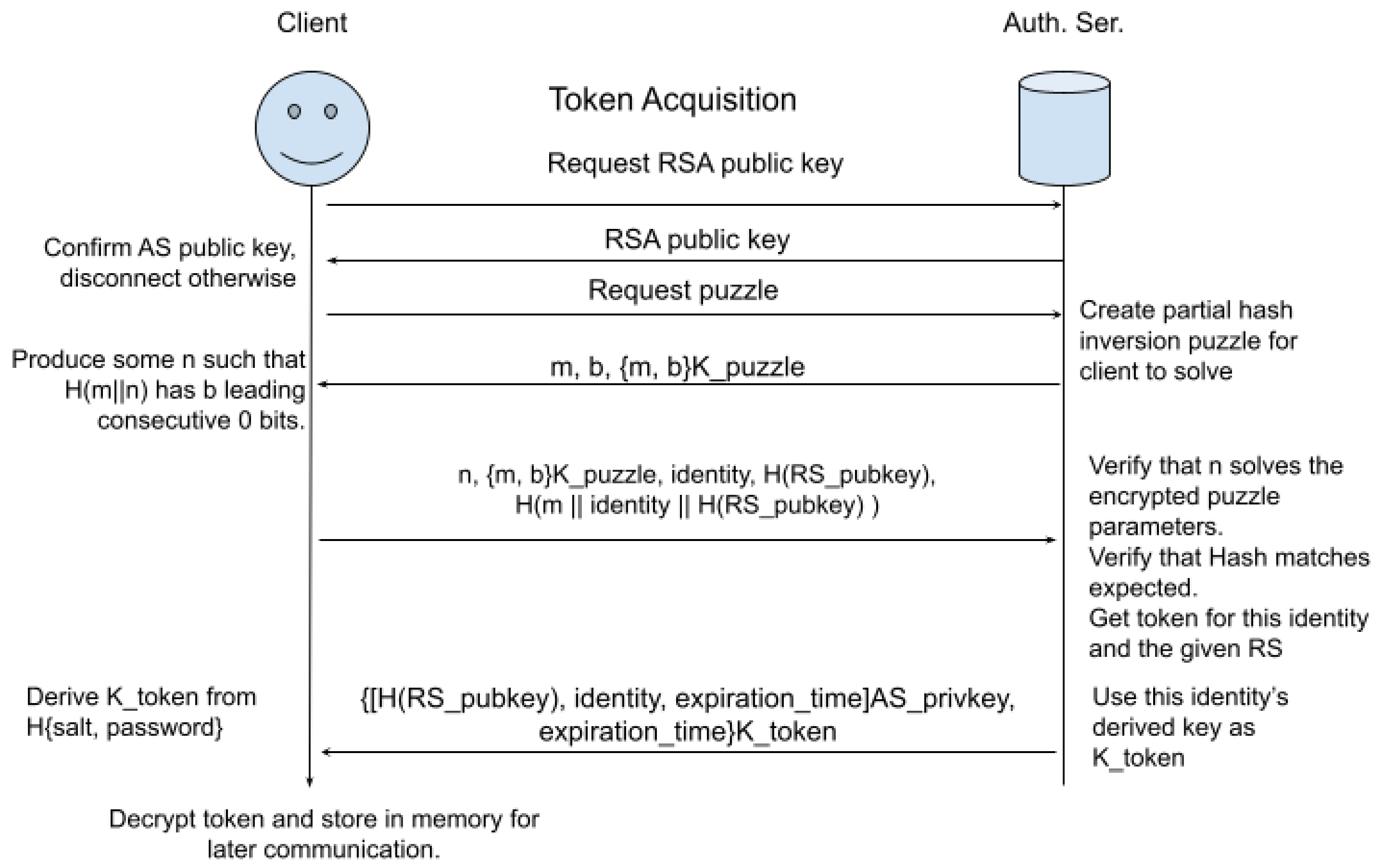
In the fifth and final phase of the project, our threat model has expanded primarily in reducing trust of the authentication server. With this change, we have to consider multiple new threats, namely that the authentication server could be impersonated or be the victim of a resource disparity denial-of-service attack, or even that the data stored on the computer running the authentication server could be collected by a malactor.

In our current leaderboard system, we have clients verify that the resource server they are connecting to is the correct server and not a possible imposter, but not for the authentication server due to its previous assumption to be fully trustworthy. Our modified threat model now means that the authentication server may also possibly be an imposter. In order to address this, we will implement a verification system similar to that of the resource server verification system. The first time that a client attempts to connect to the authentication server, the client will verify that the public key received by the authentication server is correct, which is accomplished via a visual comparison between the received public key hash and a public key hash received out-of-band. If a client has confidence they have correctly connected the first time, they may also accept the public key without comparing it with an out-of-band key, however, this is recommended against. For all future connections, the client will simply check the received public key against the stored public key for the authentication server for automatic verification, and close connection if the keys don't match. On this first connection, the user will register with the authentication server with a username and password, and this will be the only time that a password will be used; all further connections will use the salted hash of the client password to derive a symmetric key encrypting the authentication token. Limiting the password to only the initial registration with the authentication server ensures that any future attempts to connect to the authentication server will not result in compromised sensitive information if the server turns out to be an imposter or otherwise compromised.

As our current system stands, the authentication will by default attempt an RSA decryption on any packet that it receives. Now that the authentication server is under threat for a possible resource disparity attack limiting its service capabilities, this policy needs an immediate change. In order to address the threat of a denial-of-service attack using resource disparities on the authentication server, we will implement a puzzle solving system to make clients do more work than the server on each request. This system will issue any connection a partial hash inversion challenge that is to be solved by the client before any login or authentication protocol is initiated. This challenge will be easy for the authentication server to generate and verify the correct response, but will be disproportionately difficult for the client to computationally solve. Additionally, our login system will further change, as now the keys for encryption will be the previously mentioned salted password derived symmetric key, taking the burden of complex RSA computations off of the authentication server during regular logins, allowing for greater efficiency while not compromising security.

As stated previously, during the implementation of our leaderboard system we assumed that the authentication server was completely trusted and secure. Now that it is under threat like any other server used by the system, it must be acknowledged that the computer it runs on may not be entirely secure, even physically. Sensitive information is stored in plaintext on the authentication server's disk, so if someone were to gain access to the computer where the authentication server is hosted, they could potentially steal of the information and data belonging to all users, which would compromise all current accounts and require a complete remake of all users' registrations with new passwords. In order to counter this threat, The authentication server will now encrypt the database stored on disk, and will perform any database operations exclusively in memory (where it is protected by the operating system from other programs and is unrecoverable if the computer loses power), re-encrypting before being stored again on disk and clearing memory of any previous data or operations done regarding the database every time it exits memory and returns to disk.


Client
Auth. Ser.
Registration
Request RSA public key
RSA public key
Confirm AS public key, disconnect otherwise
Create partial hash inversion puzzle for client to solve
m, b, {m, b}K_puzzle
Produce some n such that H(m||n) has b leading consecutive 0 bits. Generate random K_reg
n, {m, b}K_puzzle, {K_reg}AS_pubkey, {identity, password, H(RS_pubkey)}K_reg
Verify that n solves the encrypted puzzle parameters. Register new identity and scrypt(salt, password). Get token for this identity and the given RS
Derive K_token from H{salt, password}
{[H(RS_pubkey), identity, expiration_time]AS_privkey, expiration_time}K_token, salt
Use this identity's derived key as K_token
Decrypt and store token in memory for later communication.


Client
Auth. Ser.
Token Acquisition
Request RSA public key
RSA public key
Confirm AS public key, disconnect otherwise
Request puzzle
Create partial hash inversion puzzle for client to solve
m, b, {m, b}K_puzzle
Produce some n such that H(m||n) has b leading consecutive 0 bits.
n, {m, b}K_puzzle, identity, H(RS_pubkey), H(m || identity || H(RS_pubkey))
Verify that n solves the encrypted puzzle parameters. Verify that Hash matches expected. Get token for this identity and the given RS
Derive K_token from H{salt, password}
{[H(RS_pubkey), identity, expiration_time]AS_privkey, expiration_time}K_token
Use this identity's derived key as K_token
Decrypt token and store in memory for later communication.

T8: Authentication Server Impersonation (Riley)

Threat Description

Currently, the authentication server is fully trusted to be who they say they are. This means that any attacker in the middle could claim to be the AS, as we are not checking the public key. Additionally, it should be ensured that a user will never send their password over the network to an impersonator of the authentication server. These are both in service of avoiding an attacker being able to collect a password, which would enable them to act on the victim's behalf.

Security Mechanism

To counter this threat, the solution is twofold. The first time that a user attempts to connect to the authentication server, they will verify that the requested public key hash matches the out-of-band public key hash for the authentication server. This public key will be stored on the client and will be confirmed every time the authentication server is connected to, disconnecting if the keys don't match. Once the public key has been confirmed, the user can register their identity and password with the authentication server for the first time, or request a token for a given identity and resource server pair (sending the resource server's public key). To reduce the chance that impersonation results in information leakage, the registration password will never be transmitted to the authentication server during regular logins. After registering or otherwise requesting a token, the authentication server will send the requested token encrypted with a symmetric key derived from a salted hash of the given user's password, including the salt in plaintext at the end of the message.

Argument for Efficacy

Confirming the AS public key manually ensures that the messages sent to the authentication server could only be decrypted by someone who has the complete key pair of the intended server. As long as the client has some external way of confirming the public key, an impersonator will not be able to access the password, even if the client sends the information to the wrong server.

Additionally, because the user never transmits their password after registration, any eavesdropper or imposter server will not possibly be able to gain any sensitive information. The user's identity and the resource server's public key are public information, and the password is only used locally to generate the derived symmetric key, so no password-related information is sent over the network after initial registration.

T9: Authentication Server Resource Disparity Attack (Jordan)

Threat Description

At present, the authentication server is highly vulnerable to a denial of service attack. This is because the server first performs an RSA decryption on any incoming login request, so it would be trivial for a malicious actor to exploit this resource disparity by launching nonsense packets at the server forcing it to perform thousands of RSA operations, potentially stopping the server from responding to legitimate client requests.

Security Mechanism

Our solution is twofold. First, we will require that clients solve a cryptographic puzzle in order to have their requests serviced. Second, we will alter the mechanism by which token requests are processed to no longer require the server to do as frequent RSA operations.

In order to have a request (registration or token) serviced, the requesting client will need to first request a puzzle from the server. The puzzle will be a partial hash inversion with a random 32-byte prefix message provided by the server and a number of leading 0 bits determined by the current server load. The solution will be a message n such that H(m||n) has at least b leading 0 bits. Provided to the user will be the message m, the number of bits b, and that same (m,b) pair encrypted with a symmetric key known only to the server. This key will be the same across all requests in a given day, but be randomly re-generated at midnight UTC. When a puzzle is solved by a user, they must provide the solution to the puzzle (n) alongside the encrypted (m,b) pair as proof that they received that puzzle from the server. **The server will store that prefix message and not allow it to be reused for 24 hours.**

Our login protocol will furthermore be changed to utilize password-derived keys to obtain tokens rather than RSA encryption, but we will utilize RSA for registration. In order to register, a user will encrypt their username and password using the server's public key and send that, indicating it is a registration request, along with a cryptographic puzzle solution. In order to receive a token, a user need only provide an encrypted puzzle (m,b) and its accompanying solution n, their identity, the public key of the server they wish to log in to, and the hash of (m||identity||resource server public key), all of which can be sent in plain text. As long as the server produces the same hash after decrypting m, the server will respond by sending a login token encrypted with a key derived from that user's password and a salt, which will also be provided to the user in plaintext.

Argument for Efficacy

Requiring users to find a solution to a cryptographic puzzle solves the resource disparity outright. The puzzles are simple to generate and verify for the server but require an investment of work from the client in order to have the server do any work on their behalf. By ensuring that the prefix messages are not reused in a short timespan we ensure that one solved puzzle does not allow an attacker to make multiple requests. By requiring that users include the aforementioned hash, we prevent an active attacker from capturing a user's solution and passing it off as their own.

By moving away from RSA encryption during logins we further ensure that the server's workload is kept light. Rather than decrypting multiple elements using different algorithms when a request comes in (and also encrypting the response), the server can simply look at the plaintext request and serve a response that only the requesting user should be able to decrypt.

T10: Authentication Server Database Security (Ojas)

Threat Description

Currently, there are resources in the AS's disk that we store in plaintext, such as the auth server's private key, and the usernames and passwords of the users of the service. This data can be exfiltrated maliciously, or by accident from the AS's computer. While it is impossible to remove all trust from the AS's computer, we aim to reduce this attack surface as much as possible. The new threat model now includes a malicious program that has been installed on the AS and does not have access to AS's memory.

We trust the AS's operating system to protect the AS's memory, and trust the user to not allow malicious programs to run in root (or any other user group with permissions to view memory of an arbitrary process), while the AS is running.

Security Mechanism

We now prompt the owner of the AS to input a password, and use the scrypt key derivation function to derive a key from the password. This key is used for symmetrically encrypting all currently insecure fields. The salt used for scrypt is generated randomly, and is stored in plaintext on disk. Additionally, user passwords will not be stored on disk at any point, instead using scrypt to produce a key which will be used to encrypt tokens on request later on. This means that even if the database were to be decrypted, user passwords would require a salted hash inversion to recover.

Argument for Efficacy

Given that a malicious program does not have access to the AS server process' memory, the malicious program only has access to the plaintext salt, and an encrypted password database. Since the data in the password database is symmetrically encrypted, brute force attacks on the database will fail because the password string is unpredictable and the brute forcer will not be able to detect when the correct key is used. The additional precaution of using scrypt-ed user passwords in the database means that the encryption key for requested tokens doesn't have to be recomputed for each request, and the passwords that generated those keys are astronomically difficult to recover.

Conclusion (Lukas)

With the emergence of an untrustworthy and vulnerable authentication server, our group had to protect against a variety of new attacks, namely the possibilities that the authentication server could be an imposter, fall easily vulnerable to a denial-of-service attack, or that the disk that stores the database for the authentication server may become compromised. By allowing the client to check the public key of the authentication server they are attempting to connect to against the previously stored public key, or the provided public key via out-of-band communication, it successfully prevents a user from unknowingly connecting to a malicious server (so long as the client uses best security practice). Additionally, if somehow a malicious server would be connected to the client, as long as it isn't a registration exchange, no sensitive information will be leaked as the password is no longer sent between the client and authentication server during login, and rather is used in deriving the key which encrypts login tokens.

By adding a challenge and response system, we mitigate the threat of a denial-of-service attack by ensuring that the connecting computer has to do comparatively more work, meaning sending a flurry of requests will only negatively affect the attacking computer. By switching to using the password derived keys on tokens, the server also doesn’t need to decrypt login token requests, where it once had to do an RSA decryption followed by an AES decryption for each login request.

With the implementation of an encrypted database on disk and only performing operations on the database in memory, the database content is protected from malware running on the same machine. As the database is protected by the OS while in memory, it is safer to perform tasks there. Furthermore, with user passwords being used only in key derivation, we can store the pre-derived key in place of the password itself, making it near impossible to steal the passwords of users after decrypting the database. In total, these new implementations successfully protect against an untrustworthy and vulnerable authentication server.