Group Name: securecoders

Members:
Gavin Abramowitz
Luke Charlesworth
Michael Bartlett
Owen Wurst

P3 Writeup

General Introduction

Given our project's new assumptions, we have elected to utilize a hybrid cryptographic approach to mitigate the four key threats of unauthorized token issuance, token modification/forgery, unauthorized Resource Servers, and information leakage. This approach utilizes four main cryptographic ideas as a means of security, authenticity, and integrity: (1) Public Cryptography, (2) Challenges, (3) Digital Signatures, and (4) JSON Web Tokens. During runtime, public key cryptography will be utilized to generate private-public RSA keypairs for the Resource server, and Authentication Server ***(figure 1)***. This lends itself to two major communication protocols, one for the client communicating with the Authentication Server and one for the client communicating with the Resource Server. Challenges will be used by both the Resource Server and Client in order to verify that they are who they say they are. Digital signatures will be used to verify the origin of both JSON Web Tokens and allow the user to ensure they are communicating with the real Resource server. Finally, JSON Web Tokens will allow the Resource Server to ensure it is only giving users access to resources they are permitted to access.

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.

Threat 1: Unauthorized Token Issuance

Threat Description:

The first threat in the model is that malicious users or outside entities will request the security tokens of our users. Since the permissions of this system are heavily based on these tokens, this is a major threat which would allow entities outside of a group to hijack groups to access their files, remove members, or delete the group entirely. In our current system, users supply the auth server with a username and password upon registration and login with these credentials whenever they want to access them. There are several vulnerabilities to this protocol that could allow Alice, our attacker, to realize this threat by acquiring the password and tricking the auth server into providing the security token that only Bob, our user, should have access to. Alice can do this with three types of attacks:

1. Because we are expecting a passive observer to be able to intercept communications between the client and Auth server, the threat could be realized when Alice intercepts either Bob's registration request or a login attempt and acquires his password.
2. Because we currently do not vet passwords in any way, Alice could predict Bob's password, either by guessing common passwords or by using her knowledge of Bob to deduce a password that may mean something to him.
3. Alice could continually guess random passwords until she gets one right and log in as Bob.

Solutions and Arguments:

To prevent the first attack, all transmissions between the auth server and the client, including the username and password pair that will be sent to the auth server during a login attempt, will be encrypted using a shared key. This key will be exchanged using the protocol shown in ***Figure 2*** and discussed in depth in the solutions to T4. This prevents Alice from seeing Bob's password by snooping on communications between the client and auth server. Because a new shared key is generated for every session, Alice will not be able to replay this protocol and send the encrypted password.

To prevent the second attack, Bob's initial registration attempt will be rejected if his password is not strong enough. To determine a strong password, we decided to adopt a standard 12 character minimum for passwords. In doing this, we prevent Alice from using common passwords or a knowledge of Bob to guess his password efficiently. If Bob is unable to use a common password, his password will more strongly resemble a random string.

To prevent the third attack, we will restrict the number of login attempts. Every time Bob logs in, his attempt will be timestamped, and if he has attempted to log in 3 times in the last 60 minutes, his request will be denied. This makes it infeasible for Alice to randomly guess Bob's password by maxing her out at 3 attempts per hour. We chose to set 3 as the limit for login attempts because it gives a normal person a couple chances to get their password wrong and also allows a user to log back in if they made a mistake and logged out too early.

Threat 2: Token Modification/Forgery

Threat Description:

Clients are now assumed to be untrustworthy, because of this, we can now expect users to attempt to modify and forge tokens in an effort to gain increased access rights or gain unauthorized access to the system. As a result of this new threat model, it is now necessary for servers to determine if tokens have been modified or forged and for third parties to verify if tokens were sent by a trusted authentication server.

Real life example:

ING Direct is a bank whose users fell victim to hackers who found and exploited a vulnerability using cross-site request forgery attacks. CSRs are attacks on a web server where an attacker causes an authenticated user to submit a request to a web application that the attacker is unauthorized for (essentially using legitimate tokens to make requests through an authorized client without their knowledge). In this way, attackers were able to create new accounts on behalf of a user and then transfer currency from the user to the attackers account.
For further details refer to this source:
https://people.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf

Problem With Current Implementation:

In our current implementation, we do not check for any form of authentication on any request. This means that anyone can make a request for any resource without even being logged into the system. This can be done by simply creating a HTTP request. This is insecure for obvious reasons.

Mechanism Implementation:

The mechanism chosen to protect against the modification or forgery of tokens is to implement JSON Web Tokens (JWTs) and use secure techniques to generate and validate those tokens. JWT is an open standard (RFC 7519) that defines JSON Web Tokens as JSON objects that are base64url encoded; they can also be signed, MACed, and/or encrypted. JWTs usually represent a set of claims, pieces of information about a user defining their qualities. In our case, that will be a list of the teams that the user belongs to. We will be focused on the signature aspect of the JWT for our security purposes. Additionally, to overcome the lack of communication between the Resource Server and Authentication Server, we plan to share the public key of the auth server with the Resource Server out of band for signature verification purposes, and we plan to share the public key of the Resource Server with the client as defined in ***Figure 1***. The user will store the key in a config file before running the client.

When a user logs in, they will receive a JWT from the Authentication Server to be used for the duration of their session, after which, it will expire. When the Resource Server receives a JWT from a client or when a client receives a token from the Authentication Server, they can verify the token's legitimacy by checking the signature. Because the Authentication Server always signs the token with its private key it can be verified through the public key.

We plan to use the RS256 algorithm using 4096 bit keys because it is an industry standard asymmetric solution that eliminates the problem of key sharing and allows the server or user to use the public key to verify the signature. The RS256 algorithm (RSA using SHA256) works by first hashing the header and payload of the JWT using SHA256. This protects against token modification because the recipient can always rehash the token and compare the values.

Once the token is hashed, the RSA256 algorithm is used to sign the token using a public/private key pair, ensuring anyone who receives the token can verify its original sender (RS256).

As long as every request is being checked for a token, the use of signatures during this process allows us to ensure that users are not able to increase their access level with tokens that have been tampered with or forged.

Explanation of why our implementation is sufficient:

Our proposed mechanism sufficiently addresses the threat of users modifying or forging tokens by securely generating and validating those tokens to ensure that users cannot gain unauthorized or increased levels of access. This process starts with properly generating JWTs with RS256. The security of JSON Web Tokens against modification and forgery when encoded with the RS256 algorithm is founded on two principals:

1) Digital signatures generated by public/private key pairs through the RSA256 algorithm. When signed using RSA, the integrity of the JWT's origin can be trusted because the Authentication Server signs the token using its private key. This ensures the token cannot be forged as users will not have access to the private key.

2) Hashing the header and payload of the token with the SHA256 algorithm. Any attempt to modify the token would require an attacker to rehash the token in such a way that it matches the original output. This is infeasible due to the security properties provided by the SHA256 algorithm.

Together these features guarantee that the token has not been forged or modified by a user. If illegitimate tokens are received by the Resource Server they can be easily detected in the verification process and ignored, thus ensuring security and integrity against the threat of users attempting to modify or forge tokens.

Threat 3: Unauthorized Resource Servers

Intro

The threat I am focusing on for this section is a user's (or client's) unauthorized communication with a Resource Server and the various security exploits that can be used by attackers when this occurs. This threat is best known for an attack called MITM (Man in the Middle Attack) where an attacker impersonates a given server to to intercept a communication between a user and that given server it wants to communicate with, potentially capturing sensitive information. For a more illustrative example, consider that one morning Alice decides to do some work at a local coffee shop she has been to many times before. Thomas is a hacker trying to steal Alice's private information for financial gain, so he has set up a rogue hotspot that uses a similar name and possibly some spoofing to impersonate the coffee shop's public WIFI network. When Alice powers on her laptop, it automatically connects to Thomas's rogue hotspot thinking it is the coffee shop's WIFI it previously remembered. Now Thomas is able to intercept all communication passing through his rogue hotspot, so later that morning when Alice goes to her bank's website to check her balance and enters her credentials, Thomas intercepts them. Now Thomas can forward these credentials to the bank's actual web-server so that it appears the request is coming from Alice, and get access to Alice's bank account. This example showcases the problematic nature of this threat as clever server impersonation can lead to data interception, tampering, and further unauthorized access to other services if a user participates in an unauthorized communication with a server.

Current Lack of Mechanism,

In our current application, this threat is unaccounted for and can lead to dangerous exploitations. Consider the following example ***(Figure 4)***: Since all of our endpoint requests are not encrypted and we have no mechanism for server authentication, an attacker can easily run a port scan to obtain the port the Resource Server is running on and then use some form of port spoofing on the local host to impersonate the server, intercept the request, and obtain the full request body. In the case of "/joinGroup" this means the attacker could easily obtain the group's join code, which it could then use to create an account and receive access to this group. Now the attacker would have full access to download and steal all of this group's shared "secure" resources, which in our case is a bunch of files.

Mechanism Implementation:

To provide a secure and correct method to authenticate a Resource Server and counteract this threat, first the Client must receive the Resource Server's public key out-of-band, for later verification purposes ***(figure 1)***, the client will then store the key in a configuration file so it can be utilized at runtime**.**

When the client requests to communicate with the Resource Server, the Resource Server will transmit its SHA-256 hashed public key. This will allow the client to first verify that the key hash they just received matches the Resource Server Public Key it received previously from the out-of-band key exchange. They can do this by SHA256 hashing the key they received out-of-band and comparing it to what they were just transmitted Provided the hashed values match, the client can authenticate them as *the* Resource Server and begin further communications to develop a shared private session key, and send this to the Resource Server along with their token, all encrypted with the Resource Server's public key.

Proof Security:

To prove this mechanism makes this threat unexploitable, let's consider what it would take to effectively impersonate the Resource Server now as an attacker. To do this an attacker would first need to utilize port and IP spoofing to impersonate an instance of the Resource Server, but now they would also need to pass this verification check made by the client. If they do not transmit the exact Resource Server's public key to the client, the hash values of the keys the client compares will not match, since SHA256 is both second preimage and collision resistant, and the client will identify the malicious server. However, if the attacker does obtain the public key of the Resource Server and is able to pass this verification check up until the point where client sends them their encrypted token and newly derived shared secret, the attacker would then need the Resource Server's private key to decrypt this information and steal any sensitive data. This private key is never transmitted and securely generated, thus it cannot be obtained by the attacker. Hence, this is a catch-22 situation for the attacker. In order to impersonate the Resource Server and pass initial verification they need to send the client the Resource Server's public key, but once they do they lose any chance of deriving the shared secret and decrypting future communications because they will never have the Resource Server's private key. All communications past these first two authentication steps will be encrypted with the shared AES private session key, which is also securely generated and never sent unencrypted for possible interception. Therefore, there is no feasible way a malicious Resource Server will be authenticated by the client.

Proof Correctness

The correctness of this mechanism rests on the fact that the out-of-band key exchange was initially successful, such that the client can perform this verification of the Resource Server. It is crucial that when the first connection is made the client compares the public key that the supposed Resource Server shares with what they have on file, as incorrectly trusting the Resource Server initially will compromise all future communications. However, if the client performs the specified mechanism they should be able to efficiently authenticate the real Resource Server, generate a shared session key, and transmit that key to the Resource Server, encrypted with the Resource Server's public key such that only the real Resource Server could use its private key to decrypt and obtain it. Once this occurs, both parties can be sure they have the same shared session key, as one party generated it and securely transmitted it to the other party. Thus, ensuring correctness in future communications.

Threat 4: Information Leakage via Passive Monitoring

Threat Description:

This threat means that any passive attacker has the capability to monitor the channels between our Authentication Server and client, and Resource Server and client. This threat poses a risk of sensitive information, such as a user's login credentials and their group memberships, the contents of files, the ownership of files, tokens, and any other data transmitted to and from the client being observed by the passive attacker. If this threat were to be exploited, an attacker could gain unauthorized access to files and user accounts, which would undermine the confidentiality and privacy of the system. In our current system, all data is sent as unencrypted plain text (or bytes in the case of files). This means that anyone could obtain data in transit if they are eavesdropping.

Mechanism Introduction:

In the description below, when some data is "encrypted" this means that the data is being encrypted with the specified shared key, using the implementation of AES mentioned in the *Technical Choices* section.

Mechanism Description:

This mechanism begins with an out-of-band key exchange that happens before the servers and client applications are run. The Auth Server will generate its RSA keypair and give its public keys to the Resource Server. Additionally, the client stores the Resource Server's public key as well. The Resource Server will store the Auth Server's public key in a config file and when it is run, it will output its public key to be distributed out of band to clients who will then store the key in a configuration file. Once this out-of-band key exchange takes place, the clients are ready to run. This process is illustrated in ***Figure 1.***

This protocol begins by the client entering the landing page of our application and subsequently requesting for the Authentication Server's public key. Since the Authentication Server is still entirely trusted this transmission can occur in real time. Next the client will generate a shared session AES key and send a three-part message back to the auth server; the first part will be this shared key encrypted with the Authentication server's public key, followed by the client's credentials encrypted by the shared key, and lastly a current Timestamp encrypted by the shared key. Finally, the Authentication server can receive and decrypt the shared key, which it can then use to first decrypt the timestamp and verify that this request was created and sent by a client within a 6 second time epoch. Provided this verification step passes, the Authentication Server can then decrypt the user's credentials and either register the "new user" or login an existing user. Once the user's credentials have been validated, the Authentication Server will then generate a login token, encrypting this with the shared AES session key and signed by the Auth server's public key. For all future communications in this session these parties can use this shared private session key to communicate. This process is illustrated in ***Figure 2.***

Once the client has obtained their login token, they can then communicate with the Resource Server for other requests they have while using the application. This process starts with a Resource Server verification process, in which the client requests the Resource server's public key and the Resource Server then transmits a SHA256 Hash of their public key y. The Client can then verify this public key against what they already have documented from the out-of-band key exchange. Provided these values match, the client will then proceed by encrypting and sending its login token along with another shared AES session key it generates. The Resource Server can then verify if this token was indeed signed by the Authentication Server, and if so it will then perform a Timestamp based challenge with the client, encrypting with the shared AES session key it received to verify the client was the creator of this key. After both parties mutually authenticate, they will now have a shared private key to utilize in all future communications. This process is illustrated in ***Figure 3.***

<u>How This Mechanism Sufficiently Addresses This Threat:</u>

This mechanism is able to sufficiently address this threat by ensuring that no critical information can be derived. That is, no key can be learned from the communication on the eavesdropped channels, and no important data can be read as it is securely encrypted using the derived shared keys.

Because of the out-of-band key sharing between the Authentication Server and Resource Server (specified further in *Technical Choices*), the Resource Server has obtained the Authentication Server's public key securely, and the Client has received the Resource Server's public key securely.

The keys being used are secure because, to date, there is no way to brute force these keys in a feasible manner, making it difficult for an attacker to determine any private key from a public key. Therefore, the keys are secure.

Further, when the Client first attempts to authenticate with the Authentication Server, providing a current timestamp as a crucial element to this process ensures that no attackers can intercept this authentication attempt and send it at a later date. As now the Authentication Server would easily recognize that the encrypted timestamp does not match the current time epoch, indicating a replay attack and thus detect the attacker.

Because 192-bit AES encryption is used with a shared secret, we can ensure the data being encrypted is not read as there are no known practical attacks that would allow someone without the knowledge of a key (which, as stated above, is considered secure) to access the encrypted data.

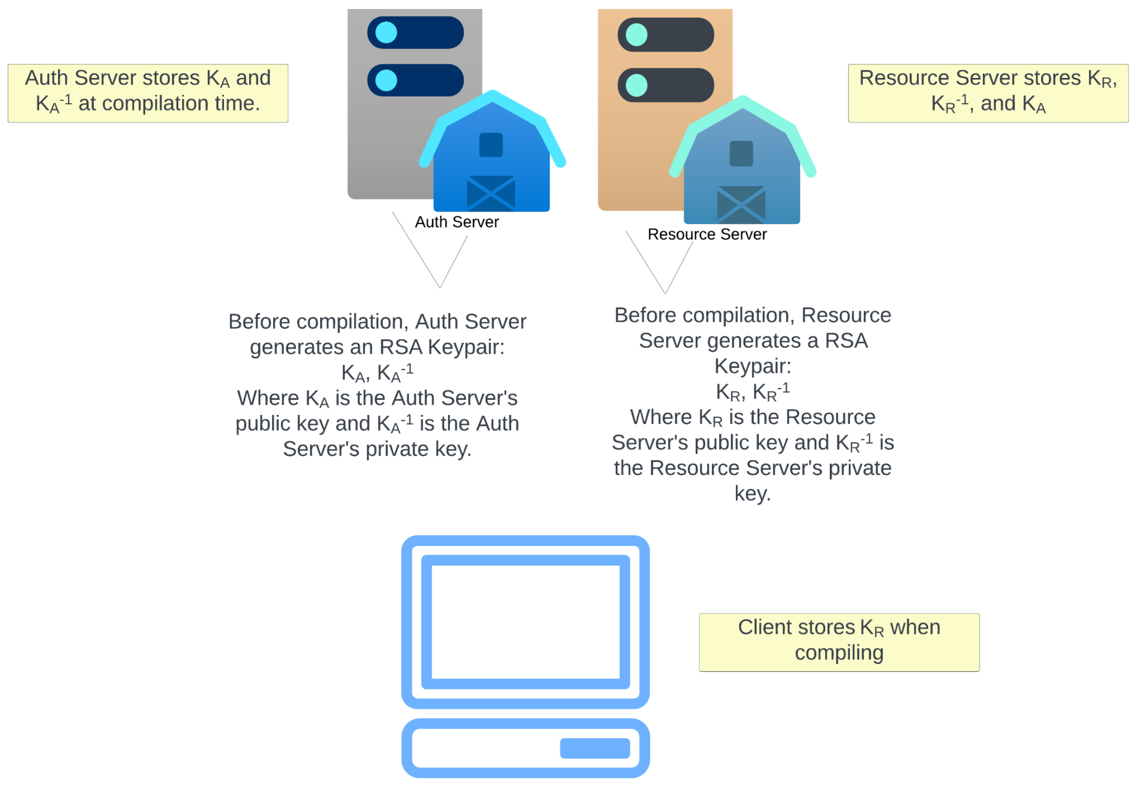
Finally, because CTR mode is being used without IV reuse, we can ensure we have a secure block cipher as CTR mode is considered secure for arbitrary length data and, plus, it is efficient for parallel processing capabilities.

Conclusion:

Each threat mitigation mechanism uses different cryptographic techniques which, to date, are known to be secure. Mitigation of threats 1 and 3 utilize the secure communication mechanism mentioned in explaining mitigation of threat 4. Since the mechanism for mitigating threat 4 is considered secure, threats 1 and 3 can use this communication protocol in order to relay the information in their mechanism securely. Our implementation for threat 2 relies on JSON Web Tokens to prove that the client is able to obtain the information that it is requesting. Because this token cannot be forged as it is signed using the Authentication Server's private key and hashed with RS256, the Resource Server can always ensure that the token did indeed come from the Authentication Server and is unmodified. As a result, all four mechanisms for mitigating the threats secure our application.

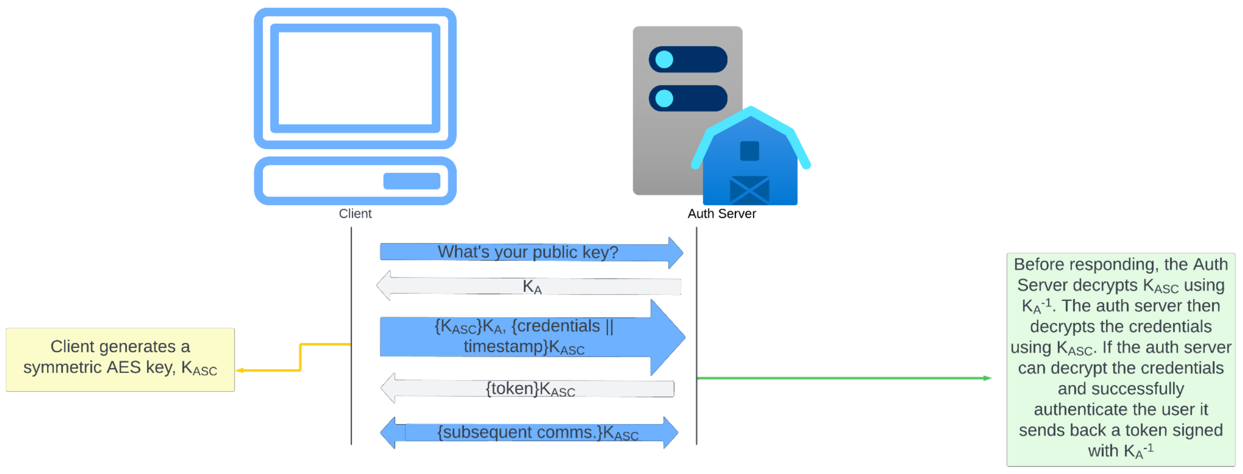
Appendix:

Figure 1: **Compilation Time Security Measures**



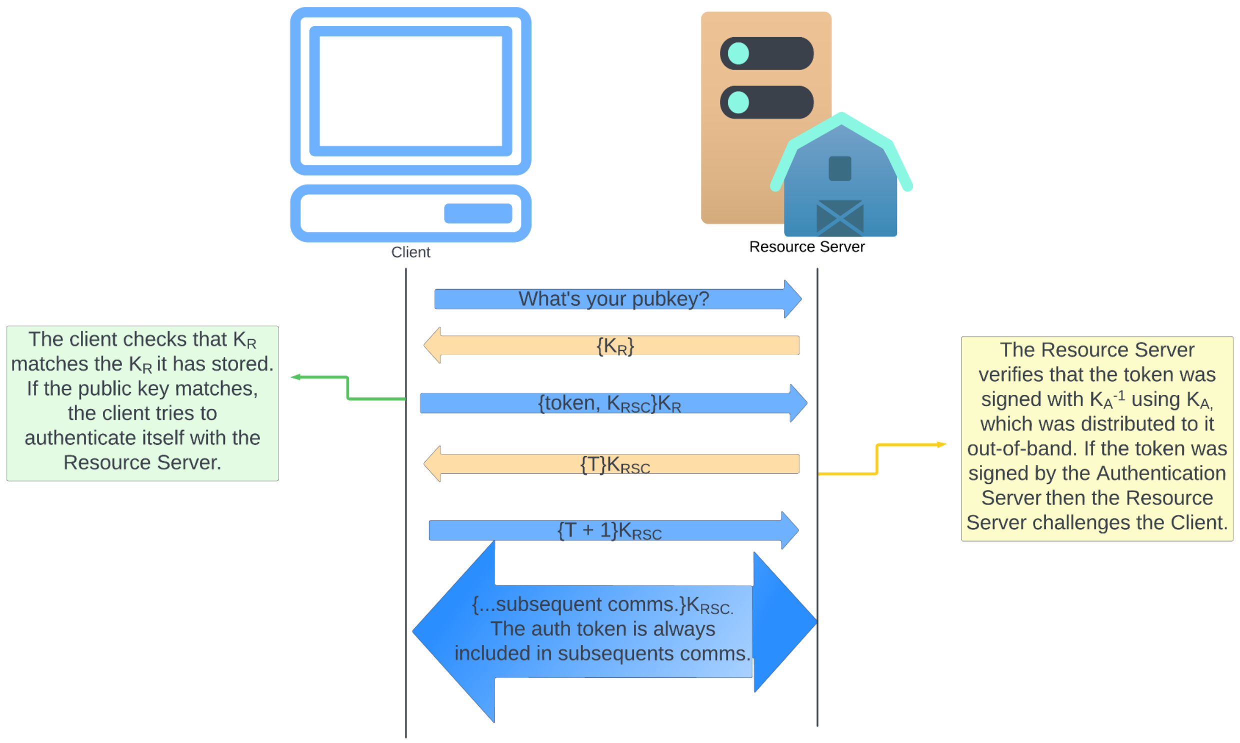
Auth Server stores K_A and K_A^{-1} at compilation time.
Auth Server
Resource Server
Resource Server stores K_R, K_R^{-1}, and K_A
Before compilation, Auth Server generates an RSA Keypair: K_A, K_A^{-1} Where K_A is the Auth Server's public key and K_A^{-1} is the Auth Server's private key.
Before compilation, Resource Server generates a RSA Keypair: K_R, K_R^{-1} Where K_R is the Resource Server's public key and K_R^{-1} is the Resource Server's private key.
Client stores K_R when compiling

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



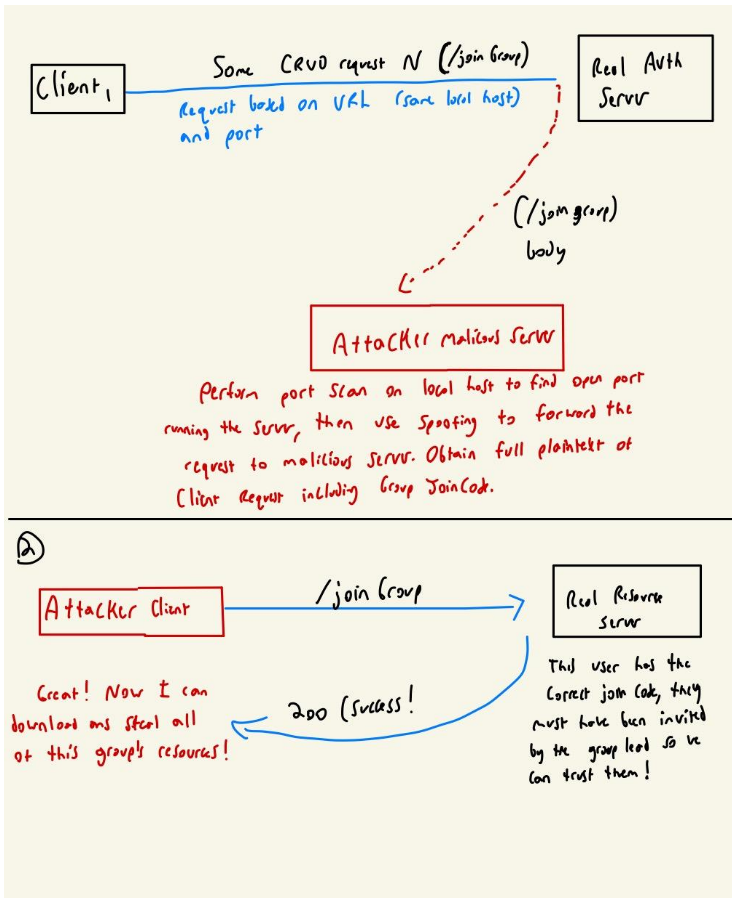
Client
Auth Server
What's your public key?
K_A
$\{K_{ASC}\}K_A$, {credentials || timestamp}K_{ASC}
{token}K_{ASC}
{subsequent comms.}K_{ASC}
Client generates a symmetric AES key, K_{ASC}
Before responding, the Auth Server decrypts K_{ASC} using K_A^{-1}. The auth server then decrypts the credentials using K_{ASC}. If the auth server can decrypt the credentials and successfully authenticate the user it sends back a token signed with K_A^{-1}

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



Client
Resource Server
What's your pubkey?
{K_R}
{token, K_{RSC}}K_R
{T}K_{RSC}
{T + 1}K_{RSC}
{...subsequent comms.}K_{RSC}. The auth token is always included in subsequents comms.
The client checks that K_R matches the K_R it has stored. If the public key matches, the client tries to authenticate itself with the Resource Server.
The Resource Server verifies that the token was signed with K_A^{-1} using K_A, which was distributed to it out-of-band. If the token was signed by the Authentication Server then the Resource Server challenges the Client.

Figure 4: **Example of Unauthorized Resource Access Exploit In Current System**



Client,
Some CRUD request w (/join Group)
Real Auth Server
Request based on URL (same local host) and port
(/join group) body
Attacker malicious Server
Perform port scan on local host to find open port running the server, then use spoofing to forward the request to malicious server. Obtain full plaintext of Client request including Group Join Code.
2
Attacker Client
/join Group
Real Resource Server
This user has the correct join code, they must have been invited by the group lead so we can trust them!
200 (success!
Great! Now I can download and steal all of this group's resources!