Secure Chat Application

Saman Jafari

Reza Mirzazade farkhani

Contents

Summary

This is the report for a secure chat application, that provides end-to-end security, mutual authentication and other security features that are discussed in rest of report.

In the following the security feature of the application, assumptions and the security algorithms used are briefly mentioned.

Security Feature

This protocol is designed to provide the below services and security features:

- Resilient against Online/offline Password Guessing
- Resilient against Server’s Database Reading
- Resilient against Reflection
- Resilient against Message Replay
- Confidentiality and Integrity protection of messages
- Mutual Authentication to be resilient for Impersonation of client and server
- Resilient against Man-In-The-Middle
- Perfect Forward Secrecy by use of Diffie-Hellman
- Resilient against shoulder-surfing while entering password
- Messages are not going through server
- Session keys between clients are not known to the server
- Ability for clients to talk without the need of the server presence

Assumptions

Clients

- Clients have the Server’s public key

- Clients have **g** and **p** for Ephemeral Diffie-Hellman pre-shared with the server

- Clients use passwords that may be weak

- values such as N for SRP and P and g for DH are pre-shared between clients and the server.

Server

- Only one session at a time is allowed for each user

- Only Server has an asymmetric private key

- Server has verifier of registered users

Cipher Suites

Security Algorithms

- Confidentiality and Integrity
- AES128+GCM
- Application: communication between client and server as well as between clients

Authentication

- HMAC+SHA256

Asymmetric Encryption

- RSA2048 + OAEP

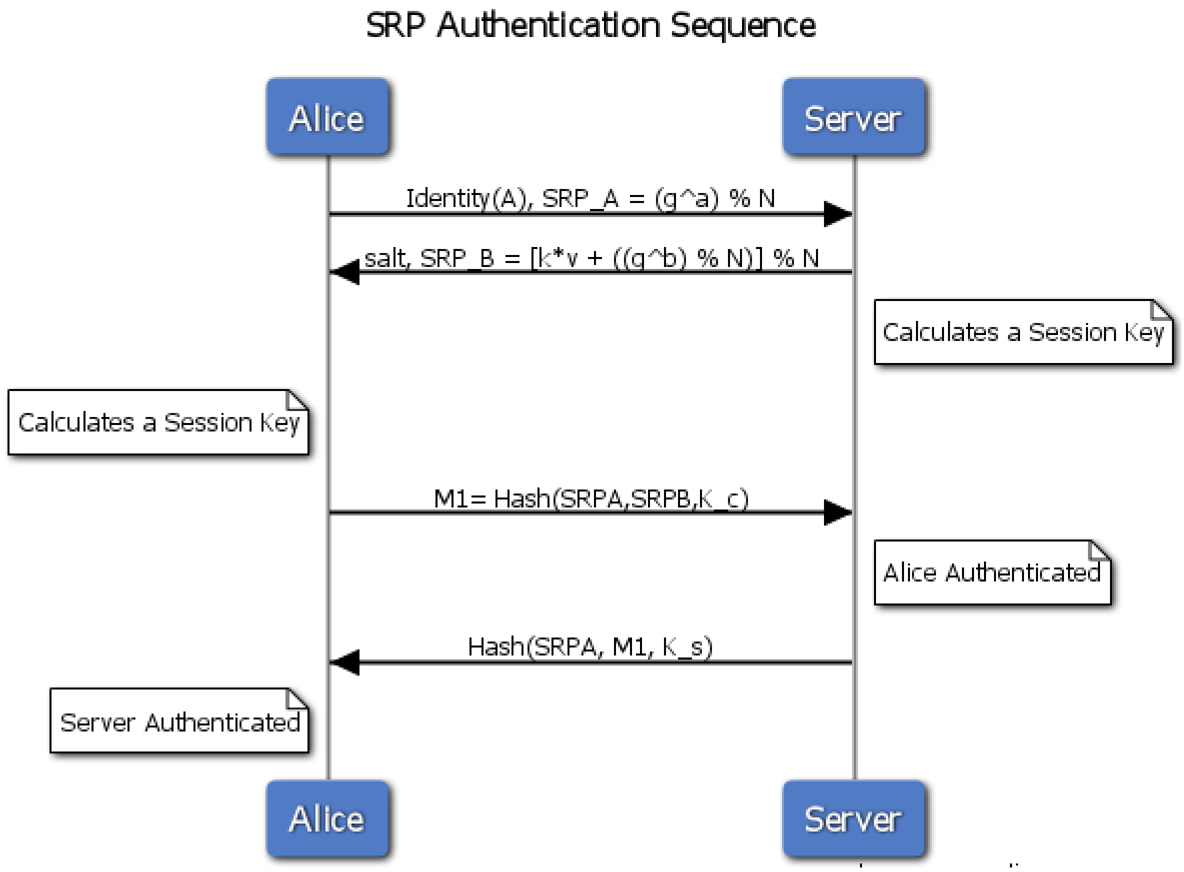
Login, Authentication and Key Exchange

SRP is used for authenticating clients to the server and vice versa. This is because of providing services as mentioned below:

- Resiliency to the server’s database reading
- Communication eavesdropping

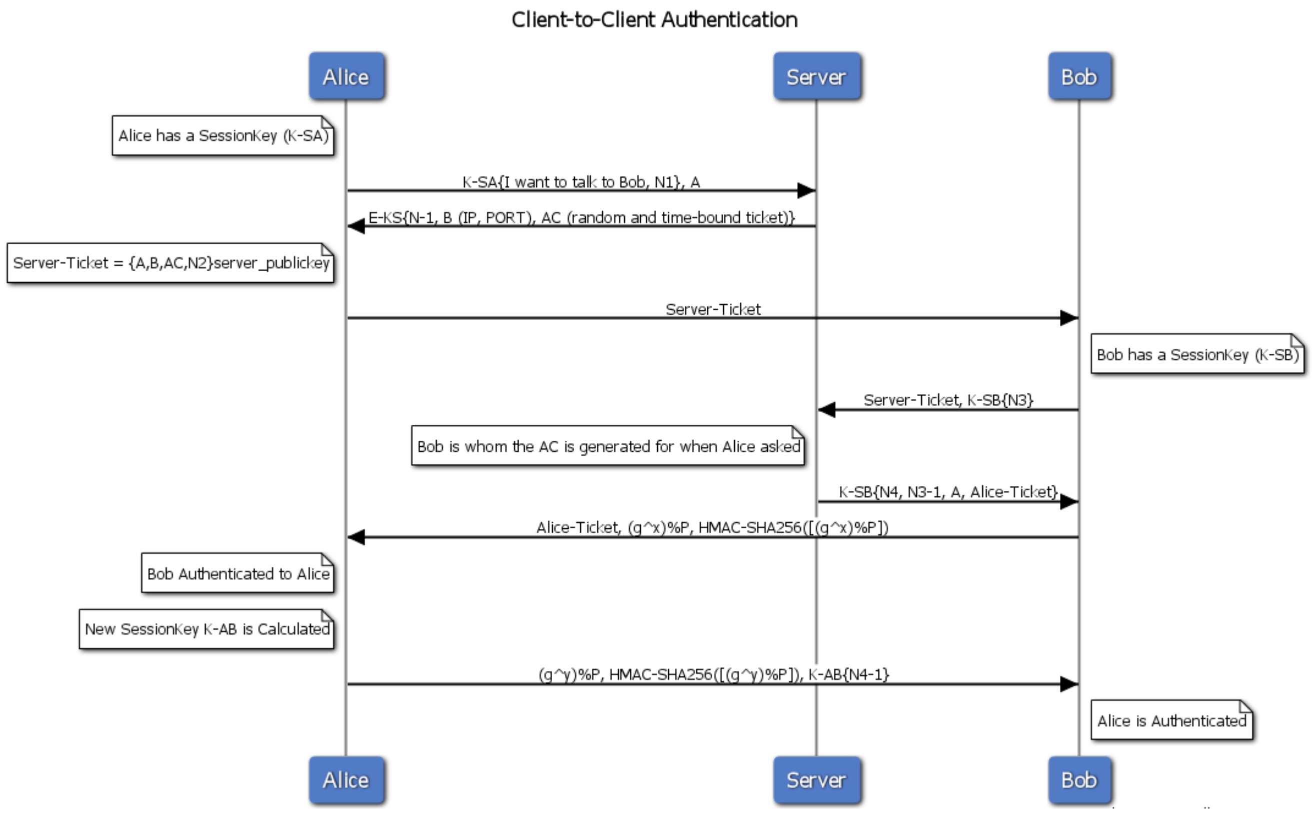
Login of client starts as the following

- User is prompted for username and password
- chatClient application sends his identity and calculated SRP values to the server
- Server calculates some values based on SRP and its stored salt too. It then sends them back to the user
- Both calculate session keys
- And finally in messages 3 and 4, the client and the server are both authenticated by sending proof that they have both calculated the expected session keys


SRP Authentication Sequence
Alice
Server
Identity(A), SRP_A = (g^a) % N
salt, SRP_B = [k*v + ((g^b) % N)] % N
Calculates a Session Key
Calculates a Session Key
M1= Hash(SRPA,SRPB,K_c)
Alice Authenticated
Hash(SRPA, M1, K_s)
Server Authenticated
Alice
Server

Client to Client Authentication and Key Exchange

When an already-authenticated client wants to talk to another authenticated client, he acquires the connection information, IP and Port, from the server by sending 'Send' command. By the use of those values, nonces and tickets the communication between clients are established using a newly-generated session key which is unknown to the server. The use of tickets and nonces guarantees the freshness of messages, prevents any Man-In-The-Middle attack, and also impersonation.


Client-to-Client Authentication
Alice
Server
Bob
Alice has a SessionKey (K-SA)
K-SA{I want to talk to Bob, N1}, A
E-KS{N-1, B (IP, PORT), AC (random and time-bound ticket)}
Server-Ticket = {A,B,AC,N2}server_publickey
Server-Ticket
Bob has a SessionKey (K-SB)
Server-Ticket, K-SB{N3}
Bob is whom the AC is generated for when Alice asked
K-SB{N4, N3-1, A, Alice-Ticket}
Alice-Ticket, (g^x)%P, HMAC-SHA256([(g^x)%P])
Bob Authenticated to Alice
New SessionKey K-AB is Calculated
(g^y)%P, HMAC-SHA256([(g^y)%P]), K-AB{N4-1}
Alice is Authenticated
Alice
Server
Bob

Step 1: First message Alice sends an encrypted request, using the session key, regarding the target's connection information to server for the IP and Port.

Step 2: Server prepare the response and encrypt it with the session key shared between the server and the client. It also sends a nonce, N1, and AC which is a randomly-generated and time-bound ticket issued specifically for Bob and Alice.

Step 3: Alice first validates the value of N1-1. If she receives the expected value she generates a ticket for server, server-ticket, which is encrypted with the server's public key and sends it to Bob.

Step 4: Bob delivers the received server-ticket to the server. In addition, he generates a nonce, N3, and encrypt it with his valid session key which is shared between him and the server.

Step 5: Server checks the validity time of AC, the value, and also it double checks whose AC is issued for. It then calculates N3-1, generates N4 and a new ticket encrypted with Alice's session key—called Alice-Ticket. It then encrypts all the values with the session key of Bob and sends it to him.

Step 6: Bob first checks the value of N3-1. Then generate a nonce, N4, and calculate his Diffie-Hellman's contribution. In order to provide authentication for the DH contribution, he generates a MAC of the DH contribution using N4 as the key for HMAC. All the values along with Alice-Ticket are sent to Alice.

Step 7: Alice decrypts the ticket received from the server, Alice-Ticket, and retrieves N4 and N2-1. She validates N2-1 and by use of N4, she validates the MAC generated by Bob. By end of this stage, Bob is authenticated to Alice as the MAC is valid. Now Alice calculates her own DH contribution, generates a MAC for it, and also calculate N4-1 encrypted with the newly-generated session key derived from DH Key Exchange.

Note: More details are given in the appendix.

Messaging Protocol

After successful establishment of key between Alice and Bob, they can simply symmetrically-encrypt their messages by the session key and send it without being worried that their communication get decrypted by anyone including the server. This is due to the fact only the Alice and Bob know the key since it is calculated using EDH, by use of private values that they only know them.

AES128_GCM is used for encryption between Alice and Bob in order to provide confidentiality and integrity.

Replay defense in messages between users

Although, an attacker cannot replay packets that are previously captured in other session conversation due to the difference in the session keys, an eavesdropper can still replay packets within the same session. For such scenario no protection has been considered since replaying of messages between two users does not impose a great risks and also observing famous chat applications such as Imo and Telegram, we come into the conclusion not to provide any replay attack defense due to the unnecessary performance overhead it has.

Replay defense in messages between users and the server

Unlike the communication between users, the messages between users and the server are sensitive to replay attack since the server is the single point of failure. Therefore, risk of replay attack is considered high in this case. Nonce is used to provide freshness of message property. So the receiver needs to return Nonce-1 in response in order to ensure that the receiver could successfully decrypted the message and also as a defense against reflection attack.

Logout Protocol

When a user logs out, a logout notification is sent to the server.

E_SessionKey{logout_message}

Replay attack in logout protocol

This attack is not possible since the session keys will be removed from the memory of the other users and the server, therefore any later replayed logout messages, will be ignored since they cannot be decrypted. For other connections, since they are encrypted with a newly-generated key, session key, therefore the captured logout message from previous conversation does not work in another conversation.

Modifications

- We use authenticated HMAC when Bob sends Diffie-Hellman parameters to Alice. In the previous design it was not authenticated and we realized there is a MITM attack here.
- In the previous design we assumed Alice wants to authenticate herself to server and send a message to bob immediately. This assumption is wrong in practice because users do not want send message after login necessarily. They also need some information of the target user. Therefore, users need to log-in first and get the list of current users of system and their information. Thus, we separated login and sending message phases.
- Removing authentication phase when Bob receives a message because he is already authenticated and does not need to authenticated himself to the server again.

Probable attacks

We assume that server is not a malicious user. If server could perform MITM attack, then he is able to calculate session key between users.

If we had more time, we would have implemented session idle timeout and absolute timeout. By so doing, server runs a scheduler to detect expired or unused sessions and remove them in order to reduce the exposure time as they are remained in the server's memory.

Discussion

Protection against weak password attack, online and offline dictionary attack

SRP proposes a strong authentication protocol in spite of the use of weak password.

Off-line Password Guessing

In order for the attacker to perform an off-line password guessing, he/she requires that a pair of clear text and the related encrypted cipher to be sent in the communication. Since there is no such correlation between a cipher and its decrypted plain text, therefore by capturing a cipher text the attacker does not know what the plain text of the captured cipher would be.

On-line Password Guessing

On-line Password Guessing is unavoidable by the attacker, however the system should develop a mechanism to be resistant against such attack. It is worth to mention that proposing solutions in this regard has cons such as consuming the server's resources and makes it susceptible to DoS attack.

We have come into conclusion to propose Slow-Down On-line Guessing attack. So that when the attacker tries to brute force/Dictionary attack a password, the system response with 1 second delay. It tremendously slowing down the on-line password brute-forcing for long and complex passwords which is true for our system.

Resiliency to DoS

While protecting against DoS completely is very difficult, in order to reduce the effect of DoS attack, we have used Nonce and AC by checking their values in the next step and preventing the messages to keep going deeper into the protocol. By so doing, less resources, time, CPU and memory is wasted since the detection happens in early stage.

Apart from that, the mechanism during login to the server is seen to slow down response time to failed authentication to makes brute forcing for the attacker far more time-consuming.

Perfect Forward Secrecy

The keys are derived from SRP or EDH. Therefore after each of which a new session key is calculated so that if this key is compromised, and also the previous messages were captured, the attacker can still not be able to decrypt the previous conversations since they were encrypted using another session key.

Resilient against shoulder-surfing

In our implementation, we considered users are typing their password in the presence of an adversary. So, passwords are not shown during the typing in terminal.

Clearing memory residue

After each authenticated communication, we remove sensitive information form memory such as AC Tickets. Thus, an adversary is not able to replay the messages and abuse the Tickets. Furthermore, session keys users are removed from memory after logout operation. Therefore, by compromising the server, an adversary cannot extract the session keys of the past communications.

If Clients do not trust Server

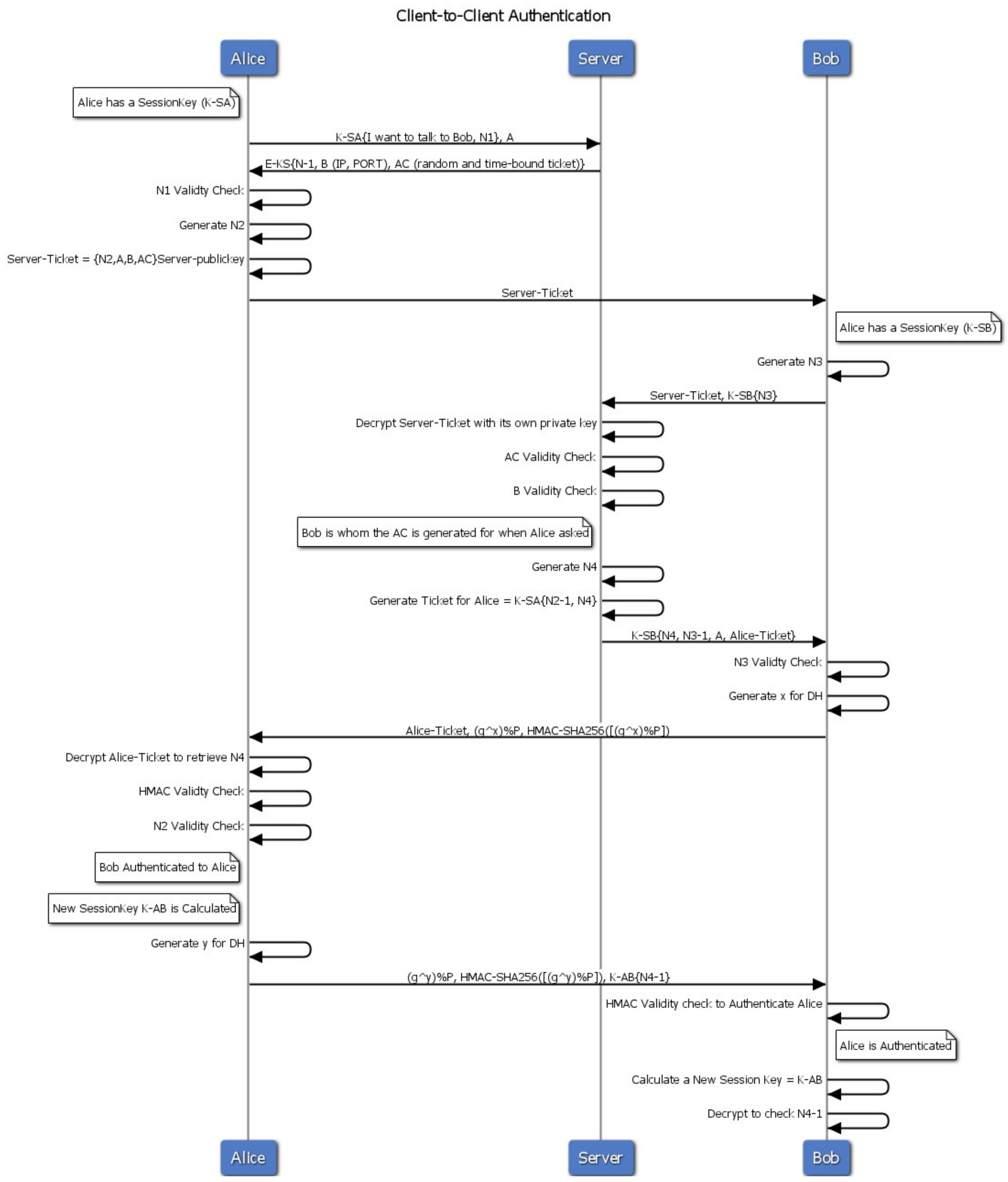
If the users do not trust the server can you devise a scheme that prevents the server from decrypting the communication between the users without requiring the users to remember more than a password?

Since session key is used between the clients that is derived using Ephemeral Diffie-Hellman, so server does not know the derived session key.

Discuss the cases when the user trusts (vs. does not trust) the application running on his workstation.

In addition, if client does not trust the Chat Application on his/her workstation, each client needs to generate an asymmetric key pair to enable the other side to encrypt the message using the sender’s public key and only the receiver of the message is able to decrypt since he/she only has access to the private key. So, for example, Alice encrypts a message using Bob’s public key, {Message}B_Publickey, and by the use of application, sends it to Bob. So Bob can only decrypt it.

Appendix


Client-to-Client Authentication
Alice
Server
Bob
Alice has a SessionKey (K-SA)
K-SA{I want to talk to Bob, N1}, A
E-KS{N-1, B (IP, PORT), AC (random and time-bound ticket)}
N1 Validty Check
Generate N2
Server-Ticket = {N2,A,B,AC}Server-publickey
Server-Ticket
Alice has a SessionKey (K-SB)
Generate N3
Server-Ticket, K-SB{N3}
Decrypt Server-Ticket with its own private key
AC Validity Check
B Validity Check
Bob is whom the AC is generated for when Alice asked
Generate N4
Generate Ticket for Alice = K-SA{N2-1, N4}
K-SB{N4, N3-1, A, Alice-Ticket}
N3 Validty Check
Generate x for DH
Alice-Ticket, (g^x)%P, HMAC-SHA256([(g^x)%P])
Decrypt Alice-Ticket to retrieve N4
HMAC Validty Check
N2 Validity Check
Bob Authenticated to Alice
New SessionKey K-AB is Calculated
Generate y for DH
(g^y)%P, HMAC-SHA256([(g^y)%P]), K-AB{N4-1}
HMAC Validity check to Authenticate Alice
Alice is Authenticated
Calculate a New Session Key = K-AB
Decrypt to check N4-1
Alice
Server
Bob

References

https://en.wikipedia.org/wiki/Secure_Remote_Password_protocol

https://cryptography.io/en/latest/