PS4: Secure Instant Messaging Application

- Tapan Singh & Paras Babbar

Assumptions:

- All users know server's public key.
- List of registered clients and online clients is stored at server.
- Server stores the password verifier, salt, User_iderntifier which every client expects if it is connecting to another client.
- User needs to remember only the password and username.
- Usernames are unique.
- First two bytes of every message is a CODE which represents the type of message.
- Users trust the server that the server cannot and/or will not decrypt the messages exchanged between the users

Architecture:

Notation	Meaning
PUB-U	RSA Public Key of User.
PRI-U	RSA Private Key of User.
N	Prime number
g	Generator for prime N. Here g = 2.
DH-PUB-U	DH Public Key of User. (g^a mod N)
DH-PRI-U	DH private key of User
DH-PUB-S	DH Public key of Server, (g^b mod N)

DH-PRI-S	DH private key of Server
PUB-S	RSA Public Key of Server.
PRI-S	RSA private Key of Server.
s	Salt
S_s, S_c	256-Bit Key for AES-GCM Shared secret generated by User (S_c) and Server (S_s).
K_{us}	256-Bit Key for AES-GCM Symmetric Key shared only between User and Server.
K_{SM}	256-bit Key for AES-GCM only known to Server.
U1	Username of USER1.
U2	Username of USER2.
IP-U1	IP Address of USER1.
IP-U2	IP Address of USER2.
DH-PUB-U1	Diffie-Hellman Public key of USER1.
DH-PRI-U1	DH Private key of User1.
DH-PUB-U2	Diffie-Hellman Public Key of USER2.
DH-PRI-U2	DH Private Key of USER2
K_{12}	256-Bit Key for AES-GCM Session Key between User1 and User2.
PORT_NO	Randomly generated port number by a user at which other users will connect.

For Testing, please use the following credentials:

1. Username: tapan
   Password: tapan311291

2. Username: paras
   Password: paras12345

3. Username: tushar
   Password: tushar12345

If the above credentials don't work or if you want to add additional users, please execute the "create_users.py" as:

"python create_users.py [username] [password]"

Note: if login fails for any user, please delete the "client_list_server.txt" file and execute the above said command to create new users. Then try to login to the server.

Message_codes = ["11", "12", "13", "14", "15", "16", "17", "18", "19", "20", "21", "22", "23", "24", "25", "26", "27", "28"]

For each client, the server will store the following details:

1. Username
2. Salt
3. Password verifier, v
   a. Here, $v = g^x \bmod n$
      And, x = H(username || password || N)
4. k = H(N || g)

These values are precomputed when the "create_users.py" is execute to create users and are stored by the server.

- The first phase is client to server authentication and key generation.
  - The process is as follows:

    (a) M = 11, time_stamp, $E_{PUB\text{-}S}${username, DH-PUB-U}

    User->Server: **M, H(M)**

(b) M = 12, time_stamp, s, DH-PUB-S

Server->User: **M, $[H(M)]_{PRI-S}$**

At user:

The client will do the following steps:

- Calculate u = H(DH-PUB-U || DH-PUB-U)
- Calculate, x = H(username || password || N)
- Calculate, k = H(N || g)
- Calculate, Shared Key, $S\_c = ((DH\text{-}PUB\text{-}S) - k * (g^x \bmod N))^{DH\text{-}PRI\text{-}U + u*x} \bmod N$
- Calculate, Symmetric Key, $K_{us} = H(S\_c)$

At Server:

The Server will do the following steps:

- Calculate u = H(DH-PUB-U || DH-PUB-U)
- Calculate, Shared Key, $S\_s = (DH\text{-}PUB\text{-}U * (v^u \bmod N))^{DH\text{-}PRI\text{-}S} \bmod N$
- Calculate, Symmetric key, $K_{us} = H(S\_s)$

(c) M_c = H((H(N) ^ H(g)) || H(username) || S_c || DH-PUB-U || DH-PUB-S || K_{us})

M = 13, time_stamp, M_c

User->Server: **M, H(M)**

(d) Server computes the above message, i.e. server calculates M_c from the values with it and compares it with the received M_c.

If verified, then Server calculates M_s

M_s = H(DH-PUB-C || M_c || K_{us})

M = 14, time_stamp, M_s

Server->User: **M, $[H(M)]_{PRI-S}$**

Now the User has been authenticated by the server.

(e) User computes M_s from the values present with it and compares it with the received M_s. If verified, the user accepts the symmetric key generated.

Now the Server has been authenticated by the User.

(f) Now User sends it's randomly generated identifier and its RSA public key to the server.

M1 = E_{Kus}{time_stamp, user_identifier, PUB-U, PORT_NO}

User -> Server: **15, iv, tag_str, tag_auth, M1**

- The second phase is user requests the server to connect to another user and start sending messages.
  - Now the User program is expecting a command, i.e. "list", "connect", "send" or "logout"
  - Command = "list"
    - If User wants to see the list of online users:

      User enters "list"

      M = E_{Kus}{ timestamp, "list"}

      User1 -> Server: **17, U1, iv, tag_string, tag_auth, M**
    - At Server:

      M = E_{Kus}{ timestamp, active_list}

      Server -> User1 : **18, iv, tag_string, tag_ath, M**

  - Command = "connect"
    - User enters "connect" and then enters the username of the User to whom to connect.
      M = E_{Kus}{time_stamp, "connect"}
      User1 -> Server : **17, U1, iv, tag_string, tag_auth, M**

      M1 = E_{Kus}{time_stamp, U2}
      User1 -> Server : **20, U1, iv, tag_string, tag_auth, M1**

    - At Server:
      M11 = E_{Kus}{time_stamp, U1, U1_identifier, PUB-U1, IP_U1, PORT-U1}
      Server -> User2 : **21, iv, tag_str, tag_auth, M11**

      M22 = E_{Kus}{time_stamp, U2, U2_identifier, PUB-U2, IP_U2, PORT-U2}
      Server -> User1 : **22, iv, tag_str, tag_auth, M22**

- Now User1 will connect and authenticate itself to User2 and generate a Symmetric key K_{12}
  M1 = E_{PUB-U2}{U1}
  M2 = E_{PUB-U2}{Client2_Identifier}
  M = 23, time_stamp, M1, M2, DH-PUB-U1, Nonce1
  User1 -> User2 = **M, $[H(M)]_{PRI-U1}$**

- At User2:
  M1 = E_{PUB-U1}{U2}
  M2 = E_{PUB-U1}{Client1_Identifier}
  M = 24, time_stamp, M1, M2, DH-PUB-U2, Nonce2, Nonce1
  User2 -> User1: **M, $[H(M)]_{PRI-U2}$**
  Now both users can generate the Shared Secret S_{12}
  Now both Users will generate the symmetric key $K_{12} = H(S_{12})$

- At User1:
  M = K_{12}{Nonce2}
  User1 -> User2: **25, time_stamp, iv, tag_string, tag_auth, M**

- Now User1 and User2 have authenticated and connected with each other.

➢ Command = “send”
- Now User1 will enter the command “send”.
  Then User1 will enter the username of the User to whon it wishes to send the message.
  Then User1 will enter the message it wishes to send.
  M1 = E_{K12}{time_stamp, U1, Message}
  User1 -> User2: **26, iv, tag_string, tag_auth, M1**
  The same is the process for User2, if it wishes to send message to User1.

➢ Command = “logout”
- **M1 = E_{Kus}{time_stamp, “logout”}**
  User1 -> Server: **17, iv, tag_string, tag_auth, M1**

  M2 = EK_{12}{time_stamp, “logout”}
  User1 -> User2: **28, iv, tag_string, tag_auth, M2**

Secure Remote Password (SRP) Protocol:

- The User remembers only password.
- SRP Protocol helps the User to authenticate to the Server.
- SRP protects against dictionary attacks and partition attacks.
- SRP has been designed to thwart the active attacks. Although it is difficult to determine conclusively whether or not these precautions bulletproof the protocol completely from all possible active attacks, SRP resists all the well-known attacks that have plagued existing authentication mechanisms, such as the Denning-Sacco attack.
- A man-in-the-middle attack, which requires an attacker to fool both sides of a legitimate conversation, cannot be carried out by an attacker who does not know User's password. An attacker who does not know x cannot fool Server into thinking he is talking to User, so at least one half of the deception fails. If the attacker doesn't know v either, he is in worse shape, because he also can't fool User into believing that he is communicating with Server.

SRP, as a verifier-based, zero-knowledge protocol resistant to dictionary attacks, offered a number of new benefits for password system implementers:

- An attacker with neither the user's password nor the server's password file cannot mount a dictionary attack on the password. Mutual authentication is achieved in this scenario.
- An attacker who captures the server's password file cannot directly compromise user-to-server authentication and gain access to the server without an expensive dictionary search.
- An attacker who compromises the server does not obtain the password from a legitimate authentication attempt.
- An attacker who captures the session key cannot use it to mount a dictionary attack on the password.
- An attacker who captures the user's password cannot use it to compromise the session keys of past sessions.

For prevention against DoS attacks, Replay attacks and Reflection attacks, we have the following protections:

- The first two bytes of every message are reserved for CODE, which tells the receiver about the type of message. Also, the time_stamps in the message helps to keep track if the messages are in order. Also, the CODE helps to keep track of each step of authentication or messaging phase. Also, User1 will always initiate the authentication process.
  This helps to protect against replay and reflection attacks.
- For prevention against DDoS attack, we may have to create multiple servers and distribute the traffic and implement ACL.

For perfect forward secrecy and end point hiding:

- We are using Diffie-Hellman key exchange to generate AES keys for encryption, so even if traffic is recorded and later analyzed, there is absolutely no way to figure out the key, even though the exchanges that may have been visible, because the key was never transmitted, never saved or made visible in plaintext anywhere.
- All the user data at the server is encrypted and saved using serve's own master key, the user data cannot be extracted even if the server data is compromised.
- The usernames are sent in encrypted format, and the server replies of information of User2 to User1 are also encrypted, which protects the identity of the Users.

If we don't trust the server to decrypt the messages exchanged between the users, then instead of sending K12 from the server, the server will send PUB-U2 and then the User1 will use this to generate a session key with User2 which cannot be derived by Server. If the User doesn't trust the application running on its machine, then we can implement a two factor authentication process, where in the User application will send out a random OTP in form of an email, and will ask the user to enter the OTP after the password. Now the application will do some predefined computation on the OTP which is sent and the OTP received and compare and if successful, the application will proceed with the login process. Also to protect the application code from being altered or reverse engineered, we can use many available code obfuscation techniques and tools to protect our complied application.