Secure IM Application

DESIGN IMPLEMENTATION

Sharad Boni, Ahmet Ozcan

Introduction

In this report, we will be describing our secure instant messaging structure with its architecture and design. First, entities that are part of the system are described with their roles. Next, we will describe all the assumptions about entities such as information that every entity has beforehand or types of keys. Then design algorithms used in order to meet security requirements of the system and implementation details such as authentication, key establishment protocols will be introduced. Last, design features for different threats will be discussed.

Entities

Our system domain includes ‘user’, ‘client’ that user interacts with and ‘server’ that enables communication between client applications.

- User
  - Only information a user has is <username,password> pair. A user is able to login from client applications on different hosts.
  - User interacts through an interface that provides these services;
    - Logging in
    - Listing active users
    - Messaging with other users
    - Logging out
- Client Application
  - Client application communicates with the server and other clients.
  - Every client has a public/private key pair.
  - It keeps a config file that includes information about the server;
    - <IP address, port number>$_{server}$
    - <Public key>$_{server}$
  - Client application does not store information about passwords of users and other information such as public key of other clients
  - Authentications are mutual, both between client←→client and client ←→ server

- Server

- Server has public/private key pair
- Public key is known by all clients
- Server keeps user informations in its database:
    - [<username,salted password hash, last session timestamp>$_{user1}$,]
- Server keeps client information:
    - [<IP address, Port number, public key>$_{client1}$,]
- Client and server authenticate mutually
- Server keeps a list of active users and notify users about the changes in the list by sending update messages.

Assumptions

- All the users have been pre-registered on the server.
- Client already has the server's IP address, port number and public key in a log file.
- Server has username ,password and a salt for each registered user stored in the database.
- After each session is complete the session key is forgotten.
- All clients have public/private key pair.
- User can login with his password from any client.
- All messages are sent with a hash. (Enc-then-MAC)
- Communication requests are sent for other online and authenticated users.

Algorithms Used

- Modified SRP-SHA256 is used for ***client(password$_{user}$)*** ←→***server*** authenication
- AES-256 with CTR mode for symmetric ancryption
- Ephemeral Diffie-Hellman Key exchange to generate session keys
- HMAC-SHA-256 for integrity check
- RSA2048 for asymmetric encryption

Implementation

- Authentication

Registered Users

- During registration, server keeps: username, password verifier, salt information

```
<salt> = random()
x = SHA(<salt> | SHA(<username> | ":" | <raw password>))
<password verifier> = v = g^x mod(N)
```

- Authentication of user is performed by a **‘password’**.Server keeps a **password verifier** that is generated from salted password hash.
- Server also keeps track of the duration that the **salt**(per user) has been used for. Salt lives for 3 months periods.

DB ENTRY → <username,password verifier(v), salt , salt_creation_date>$_{user\_i}$

SRP-SHA

- Password authentication uses SRP implementation.
- Messages are encrypted with boths side’s public keys.
  We didn’t encrypt the messages in this step because it is not required as described in the SRP protocol.We could have but it was giving implement implementational errors.

N	→ safe prime
g	→ generator for N
a,b	→ random session secrets
A,B	→ random session public keys
k	→ *H*(*N*, *g*) for SRP
u	→ *H*(*A*, *B*)
PK(C/S)	→ public keys

C **S**

C generates random *a*

a = random()

A = g^a mod(N)

k = H(N, g)

→ {**username, A**}$_{PKS}$→

← {**puzzle**}$_{PKC}$←

→ {**puzzle answer**}$_{PKS}$→

S generates b,

v ← from database

b = random()

k = H(N, g)

B = (vk + g^b)

mod(N)

← {**salt, B**}$_{PKC}$←

x = SHA(salt | SHA(username | ":" |<raw password>))

u = SHA(A, B)

SessionKey=SHA($(B - kg^x)^{(a + u * x)}$ mod(N))

u = SHA(A, B)

SessionKey=SHA($(A*v^u)^b$ mod(N))

$= SHA((g^b)^{(a + u * x)})$ $= SHA((g^b)^{(a + u * x)}$
)

$M_1 = SHA (H(g) | H(U) | s | A | B | K)$

$\rightarrow \{M_1\}_{PKS} \rightarrow$

$M_2 = SHA(H(A | M | K))$

$\leftarrow \{M_2\}_{PKC} \leftarrow$

(verification of key)

User Commands

System will have different message types that are sent in JSON format. Message types include:

- Login
- List
- Send User Message
- Logout

These commands will be encrypted and hashed with session keys between users and the server.

User to User Communication

- **Communication Request**

User A has the list for online users with ***>list*** command. When A wants to talk with B, address and public key information is requested from the server.

C_A S

$\rightarrow \{C_A,C_B\}_{SessionKey\_A\text{-}S} \rightarrow$

$\leftarrow \{C_A,C_B,\text{<ip,port,public key>}_B\}_{SessionKey}$

Instead of sending <ip,port> here we are sending these when the client successfully logs in and then we are sending the update commands whenever a new user logs in

Server notifies user B and sends the public key of A

S C_B

$\rightarrow \{C_A,C_B\text{'<ip,port,public key>}_A\}_{SessionKey\_B\text{-}S} \rightarrow$

- **Key Establishment**

For each session a key between users is established using Diffie-Hellman with ephemeral values which has been described below:

The most basic implementation of this protocol uses p and g where p is a prime and g is a primitive root modulo p. Both the values ,p and g, are chosen in a way such that the resulting shared secret can take on any value from 1 to p–1.

The Diffie-Hellman credentials are sent to the other party by encrypting it with the private key of the sender for trustability reasons and to counter man in the middle attacks.

A nonce is used as session id while creating the session keys.Usage of timestamps has been implemented. These keys are forgotten after the session gets over.

- **Messaging**

Each message is AES encrypted with the original message and the sender's identity. Also this is concatenated with the HMAC of the original message and the sender's identity.

$A \rightarrow B$: K_{AB} {A, message }, HMAC{A, message}
$B \rightarrow A$: K_{AB} {B, message }, HMAC{B, message}

- Log out

Active User List

- Server keeps a list of active users. Whenever there is an update in the user list with either a login or logout, the server updates the list
- When a user logs out, server sends an updated list to all active users

C **S**

$\rightarrow \{\text{Logout}\}_{\text{SessionKey}} \rightarrow$
$\leftarrow \{\text{ACK}\}_{\text{SessionKey}} \leftarrow$
(Both side deletes the information about session)

- Server updates active user list and send it to all users.

Protections

- **Weak password**: Hash cracking techniques like lookup tables and rainbow tables work because each and every password would be hashed the same way. We can fiddle with the hash technique such that when two same passwords are hashed both would map to a different hash end result.This technique is called salting.We can apply by appending or prepending the password with the salt string and then hashing the result.We have to store the salt string also with the hash itself.
- **Server Trustability**: Both client and server decide the key for each session mutually without the involvement of the server and also they send these messages with the public key of the receiver so the server cannot decrypt the traffic.
- **Dos:** A hash puzzle that is already known by server is sent to client to keep him busy. Verification is only a table lookup.
- **Perfect Forward Secrecy:** Keys are created with ephemeral values for both authentications and key exchanges. DH parameters are created and used only for current session.
- **Replay:** While A tries to talk with B, sessionid/nonce from B for key creation prevents A's messages to be replayed.
  we are sending timestamps with each message and if the timestamp is more than a specified amount of time we will discard those messages.
- **End-point-hiding:** Messages are sent encrypted throughout all session. No message will be sent in the clear. Starting with authentication's first message, it is encrypted with public key of server.