HIGH LEVEL DESIGN

SECURE INSTANT MESSENGER

By Husky Hackers (Group # 7)
Abdulla Al Ali
Deepak Kalra
Raghuram Krishnamachari
Shinn Chyang
1. INTRODUCTION

This document is a proposed design of a SECURE INSTANT MESSENGER. After we look at the goals and use cases, we will focus on the message exchange protocols which are a combination of symmetric, asymmetric, hashing functions. The protocols are designed keeping in mind that the user only remembers a single password. In addition, a trivial implementation of the secure instant messenger will be demonstrated.

1.1 DESIGN GOALS

The main feature of the protocol is to prevent an adversary to read / intercept the instant messages so as to retain the privacy between the users of the application. In addition to this, the following goals have been identified:

- Prevent Server to decrypt the p2p message.
- Allow clients pairs to choose their session keys
- Mutual Authentication between Server – Client
- Mutual Authentication between Client – Client
- Session key establishment for communication between Server-Client & Client-Client
- Perfect Forward Secrecy
- Protect against week passwords
- Resistance to Denial-of-Service Attacks
1.2 ARCHITECTURE

The model of our system is described as the diagram below. The server is in charge to authenticate all user(s) in the system and will allow only those users to use its resource. Once a user is authenticated to a server, it will share a unique secret key which ensure secure communication with a server.

To establish a communication with another user, both users need to be authenticated to the server first. The user, who initiates the conversation with another user, needs to get a TICKET from the server first. The user will use this TICKET for the authentication process to the recipient user. If mutual authentication is established between users, they will establish a unique session key to encrypt their conversation.

1&2) Mutual authentication with server
3) User obtains ticket from server
4) Mutual authentication with client using the ticket
2. USE CASES

2.1 LOGIN

<table>
<thead>
<tr>
<th>Description</th>
<th>User Login with username and password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>User, server</td>
</tr>
<tr>
<td>Assumption</td>
<td>The user has an existing record with the server</td>
</tr>
</tbody>
</table>
| Steps                  | 1) User sends LOGIN request to the server  
                            2) Server responds back with a challenge C  
                            3) User sends login credentials along with C  
                            4) IF Challenge is correct THEN  3.1 Server verifies password  
                                ELSE  3.2 Abort Login  
                            5) IF password correct THEN  3.1 User Login successful  
                                ELSE  3.2 Abort Login |
## 2.2 LIST

<table>
<thead>
<tr>
<th>Description</th>
<th>Fetching the list of users online</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>User, server</td>
</tr>
<tr>
<td>Assumption</td>
<td>The user has already been authenticated with IM server</td>
</tr>
</tbody>
</table>
| Steps                     | 1) Client sends a LIST request to the server with timestamp  
                             | 2) Server verifies timestamp and sends list of [usernames] |

## 2.3 PERMIT

<table>
<thead>
<tr>
<th>Description</th>
<th>Get a TICKET to chat with user B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>User, server</td>
</tr>
<tr>
<td>Assumption</td>
<td>The user has already been authenticated with IM server</td>
</tr>
</tbody>
</table>
| Steps                     | 1) Client sends a PERMIT request for user B  
                             | 2) Server verifies timestamp and sends TICKET-to-B |

## 2.4 PEER – PEER AUTHENTICATION

<table>
<thead>
<tr>
<th>Description</th>
<th>Begin conversation with user B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>User A, User B</td>
</tr>
<tr>
<td>Assumption</td>
<td>User A has obtained a TICKET-to-B</td>
</tr>
</tbody>
</table>
| Steps                     | 1) User A sends HELLO and TICKET-to-B  
                             | 2) IF Ticket valid THEN 2.1 Respond timestamp challenge  
                             | ELSE 2.2 Reject Ticket |

## 2.5 PEER – PEER SECURE CHAT

<table>
<thead>
<tr>
<th>Description</th>
<th>User message exchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>User A, User B</td>
</tr>
<tr>
<td>Assumption</td>
<td>User A &amp; B are mutually authenticated</td>
</tr>
</tbody>
</table>
| Steps                     | 1) Either User A or B can send a message with timestamp  
                             | 2) IF timestamp valid THEN 2.1 Display Message  
                             | ELSE 2.2 Discard Message |
3. PROTOCOLS

In this section we will look at our cryptographic functions in detail. In order to avoid the server administrator to eavesdrop into peer-peer communication, we have designed the protocol based on authentication message exchange between the clients, to establish the session key. Before we look at the protocols in detail, let’s look at some of the abbreviations used in the flow messages.

Key / Abbreviations

UA: Username of client A
UB: Username of client B
IP_A: IP address of client A
IP_B: IP address of client B
PK_A: Public Key of client A
PK_B: Public Key of client B
PK_s: Public key of Server
T: Timestamp

NOTE: The clients and the server must be loosely time synchronized with a maximum time delta Δt that is configurable (by default it is 120 seconds).

3.1 LOGIN

Pre-Condition: The user runs the client executable.
Action: User enters username and password in the textboxes and clicks “Login”

<table>
<thead>
<tr>
<th>REQUEST ID</th>
<th>FLOW</th>
<th>DETAILS</th>
<th>ENCRYPTION TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID_210</td>
<td>A -&gt; S</td>
<td>LOGIN</td>
<td>PLAINTEXT</td>
</tr>
<tr>
<td>RID_220</td>
<td>S -&gt; A</td>
<td>C</td>
<td>PLAINTEXT</td>
</tr>
<tr>
<td>RID_230</td>
<td>A -&gt; S</td>
<td>C, {T_1, UA, PK_A, h(pwd)PK_s}</td>
<td>RSA</td>
</tr>
<tr>
<td>RID_240</td>
<td>S -&gt; A</td>
<td>{UA, T_1, T_2, KA}PK_A</td>
<td>RSA</td>
</tr>
<tr>
<td>RID_250</td>
<td>A -&gt; S</td>
<td>KA {T_2}</td>
<td>RSA</td>
</tr>
</tbody>
</table>

RID_210: The client sends a LOGIN message in clear text.
RID_220: Server responds with challenge C. C is stateless cookie used to avoid DoS attacks.
RID_230: Along with C, user sends the login credentials (username, h(pwd), and public key). This message also contains the timestamp T_1. This is added for two reasons, one for the server to verify the freshness of the message, and two, used as a challenge for the server.
RID_240: Server verifies the timestamp T_1 and then creates a new session key KA and send it back to the client encrypted with A’s public key received in the previous message.
RID_250: Once the client responds to challenge T_2, the user is logged in.
3.2 LIST

Pre-Condition: The user has to be logged-in.
Action: User clicks on “List”

<table>
<thead>
<tr>
<th>REQUEST ID</th>
<th>FLOW</th>
<th>DETAILS</th>
<th>ENCRYPTION TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID_310</td>
<td>A -&gt; S</td>
<td>LIST, $U_A \cdot K_A{U_A', T_1}$</td>
<td>AES</td>
</tr>
<tr>
<td>RID_320</td>
<td>S -&gt; A</td>
<td>$K_A{[T_1, [usernames]]}$</td>
<td>AES</td>
</tr>
</tbody>
</table>

RID_310: Client sends its username and a challenge $T_1$ encrypted with the newly established session key $K_A$.
RID_320: The server responds to the challenge and an array of user details of all the online users. After receiving the list of online users, the client updates the display.

3.3 PERMIT

Pre-Condition: Client has a list of online users.
Action: The user double clicks on a username.

<table>
<thead>
<tr>
<th>REQ ID</th>
<th>FLOW</th>
<th>DETAILS</th>
<th>ENCR TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID_410</td>
<td>A -&gt; S</td>
<td>PERMIT, $U_A', K_A{[T_1', U_A', U_B]}$</td>
<td>AES</td>
</tr>
<tr>
<td>RID_420</td>
<td>S -&gt; A</td>
<td>$K_A{[T_1', U_A', U_B', IP_B', PK_B]}, TICKET-to-B$</td>
<td>AES</td>
</tr>
</tbody>
</table>

TICKET-to-B = $K_B{[T_T', U_A', U_B', IP_A', PK_A]}$

To initiate a chat with user B, user A needs to know the IP address and public key of B. Also, in our design the server issues a TICKET-to-B.
RID_410: Client A sends a PERMIT to the server requesting it to create a TICKET-to-B.
RID_420: Server now verifies the timestamp $T_1$ and ensures both $U_A$ and $U_B$ are online. If everything is good, the server creates a TICKET-to-B encrypted with B’s session key $K_B$. Note that the ticket has a timestamp $T_T$, user A stores this and expects B to send this value back when user A & B are trying to mutually authenticate.
3.4 PEER-PEER AUTHENTICATION

**Pre-Condition:** Client has obtained the TICKET-to-B and knows IP_B & PK_B

<table>
<thead>
<tr>
<th>REQUEST ID</th>
<th>FLOW</th>
<th>DETAILS</th>
<th>ENCRYPTION TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID_510</td>
<td>A -&gt; B</td>
<td>HELLO, TICKET-to-B</td>
<td>PLAINTEXT/AES</td>
</tr>
<tr>
<td>RID_520</td>
<td>B -&gt; A</td>
<td>{U_A', U_B', T_f, T_1, K_{AB} }PK_A</td>
<td>RSA</td>
</tr>
<tr>
<td>RID_530</td>
<td>A -&gt; B</td>
<td>K_{AB} {T_1}</td>
<td>RSA</td>
</tr>
</tbody>
</table>

**RID_510:** User A sends a HELLO along with TICKET-to-B

**RID_520:** User B decrypts the TICKET received and verifies timestamp T_f. This T_f is sent back to A. This is how A authenticates B. And for B to authenticate A, a new timestamp T_1 is added to this message. Note, B selects a new session key K_{AB}, this is sent in the same message.

**RID_530:** Users A & B are mutually authenticated when A responds to challenge T_1.

3.5 PEER-PEER MESSAGE EXCHANGE

**Pre-Condition:** Client A and Client B have been mutually authenticated and the previous chat message has successfully been acknowledged.

**Action:** Client A / B types another message and hits enter.

<table>
<thead>
<tr>
<th>REQUEST ID</th>
<th>FLOW</th>
<th>DETAILS</th>
<th>ENCRYPTION TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID_610</td>
<td>A -&gt; B</td>
<td>U_A', K_{AB} {T_1, message1}, h(message1)</td>
<td>AES + SHA1</td>
</tr>
<tr>
<td>RID_620</td>
<td>B -&gt; A</td>
<td>K_{AB} {T_1}</td>
<td>AES</td>
</tr>
</tbody>
</table>

**RID_610:** Let’s say A wants to send a message to B. A encrypts the message with the new session key established K_{AB}. Along with this hash of the message is also sent to verify the integrity of the message at the receiver.

**RID_620:** B verifies the freshness of the message. If T_1 is within the time skew, the message is displayed, and responds to A. Message is discarded if T_1 is not within the time skew.
3.6 LOGOUT

**Pre-Condition:** Client A is logged in.

**Action:** Client A clicks on “Logout”.

<table>
<thead>
<tr>
<th>REQUEST ID</th>
<th>FLOW</th>
<th>DETAILS</th>
<th>ENCRYPTION TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID_710</td>
<td>A -&gt; S</td>
<td>LOGOUT, $U_A$, $K_A{U_A', T_1}$</td>
<td>AES</td>
</tr>
<tr>
<td>RID_720</td>
<td>S -&gt; A</td>
<td>$K_A{T_1}$</td>
<td>AES</td>
</tr>
</tbody>
</table>

**RID_710:** Client A sends the logout message with the timestamp.

**RID_720:** The server marks client A as offline and forgets the session key after it responds with challenge $T_1$.

4. IMPLEMENTATION

Our design implements a **finite state machine**, and the diagrams below illustrate the client and server behavior based on specific actions:
SERVER

RID_240
Wait for client to respond to challenge T2
Start 3 sec timer

RID_220
Waiting for client to respond to challenge C along with login credentials
Start 3 sec timer

RECEIVED RID_210 (LOGIN)
SEND RID_220

TIMEOUT OR INCORRECT PWD

CORRECT C
CORRECT PWD
SEND RID_240

CORRECT T2
SUCCESSFUL LOGIN

TIMEOUT
LOGIN FAILED

SERVER
STEADY STATE

RECEIVE RID_310
(LIST)

WRONG T1

RID_320
Verify user UA online.
Verify T1 within skew.

CORRECT T1
SEND RID_320
[ usernames]

CORRECT T1
SEND RID_420
(TICKET-TO-B)

RID_420
Verify user UA online.
Verify T1 within skew.

WRONG T1

RECEIVE RID_410 (PERMIT)