LMMT-SecureDove

CPTS 428 Fall 2023
Professor Haipeng Cai

Logan Kloft
Manjesh Puram
Matthew Gerola
Taylor West

Table of Contents

Updated Software Design

In the previous report, Deliverable 3, we described three areas that our application lacked security in: key sharing, profile creation, and message resiliency. This section of the report describes the changes to the software to shore up these three shortcomings.

1. Key Sharing

In the previous software version, there was a vulnerability where the symmetric key was visible without being decrypted. The design has been updated such that clients now send a public key when creating a new profile; the server uses this public key to encrypt the symmetric key before sending it to the client, who decrypts the symmetric key using their private key. The program uses the PKCS#1 cipher with v1.5 padding. The RSA key is 2048 bits.

2. Profile Creation

In the previous software version, users could join a chat room with no name or mimic the names of other users. In the case of the former, a user can spy on a chat room, in the latter case, a user may pose as that of another user. In the new design, the client ensures that no blank username is submitted to the server. Meanwhile, the server verifies minimum and maximum length requirements, uniqueness criteria, and alphabet criteria. Additionally, a user may attempt to log in multiple times without disconnecting from the socket; in the previous version, a login attempt would always succeed, and there was no mechanism for retrying.

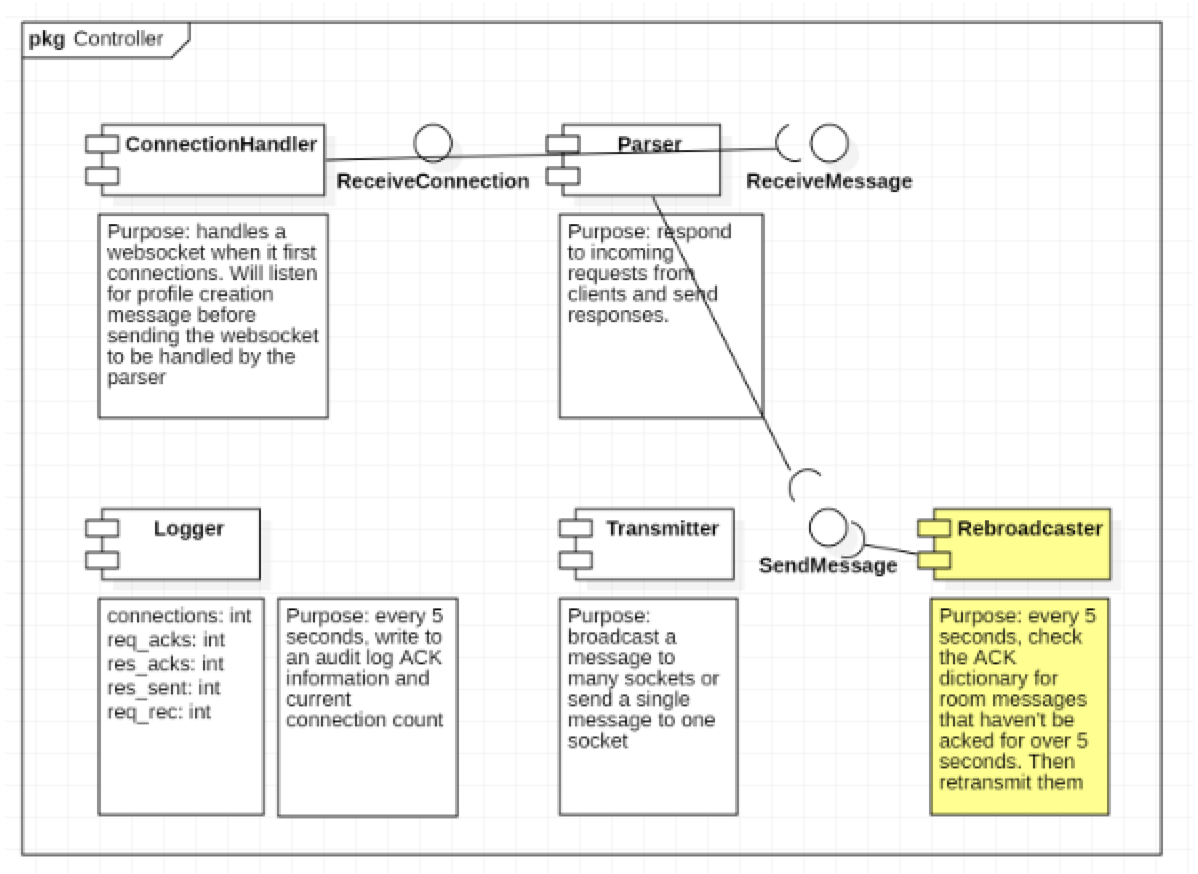
- Username must be a string
- Username must be between 1 and 32 characters inclusive
- Username can only be alphanumeric (0-9) or (a-z) or (A-Z)
- Username must not already belong to an active user

3. Message Resiliency

In the previous software version, dropped chat messages were not rebroadcasted, thus, users in a chat room might miss messages from other users in the room. The new design only targets chat messages and rebroadcast messages to individual clients who never acknowledged receiving a chat message they were supposed to.

- Messages are sent per-user basis
- Messages are re-sent at a minimum every 5 seconds with no maximum limit on latency.

The new design above introduced a new component to the controller component of our architecture and changed the data model of the application. In the controller subsystem, we see the addition of a rebroadcaster; this is the component that controls rebroadcasting chat room messages to users who never acknowledged the original chat message they received. The updated controller subsystem looks as follows:


pkg Controller
ConnectionHandler
ReceiveConnection
Parser
ReceiveMessage
Purpose: handles a websocket when it first connections. Will listen for profile creation message before sending the websocket to be handled by the parser
Purpose: respond to incoming requests from clients and send responses.
Logger
Transmitter
SendMessage
Rebroadcaster
connections: int
req_acks: int
res_acks: int
res_sent: int
req_rec: int
Purpose: every 5 seconds, write to an audit log ACK information and current connection count
Purpose: broadcast a message to many sockets or send a single message to one socket
Purpose: every 5 seconds, check the ACK dictionary for room messages that haven't be acked for over 5 seconds. Then retransmit them

You can see the added component highlighted. The Rebroadcaster component uses the transmitter's facilities to rebroadcast messages to individual users. As mentioned, the data model also changed. The data model had to change to allow the rebroadcaster to track when messages were sent, who the message belonged to, and whether rebroadcasting the message should use the encrypted or unencrypted send method. The below shows the additions to all data models that use any version of the send or broadcast commands:

```
{
        “type”: “...”,
        “verb”: “...”,
        “content”: {...},
// original data model information represented by ‘...’

// new data model information below:
        “userid”: “uuidv4”,
        “ack_timestamp”: number,
        “useEncryptedSend”: True|False
}
```

Validation Process

In the Validation Process Section, we describe how we verify that the updated software design successfully patches the identified security vulnerabilities from Deliverable 3.

1. Key Sharing

To verify the changes to the key sharing process, we observe the network communication from server to client (since the server sends the symmetric key), we look to see whether the symmetric key is successfully encrypted or not. To verify this, we include both the unencrypted symmetric key and the encrypted symmetric key to compare whether they are different. For proof of actually decrypting and using the decrypted symmetric key, see the login.jsx file starting at line 55.

2. Profile Creation

To verify the changes to profile creation, we test each criterion separately. To test uniqueness, we create two clients and attempt to sign in as “Logan” on both. To test username length, we create a user with no username and a username with a length of 33. To test the alphabet, we attempt to sign in with a username that contains a space and a separate username that contains ‘#.’ In all cases, we expect the user not to be able to sign in; in the case of the two-user test, one should be able to sign in, and the other should not.

3. Message Resiliency

To verify the changes to message resiliency, we will drop every other message for the clients. After five seconds, we should see a new message for both clients who dropped 50% of their received messages. The test will consist of two users inside of a chat room; one user will send two messages immediately, both messages should be visible to both users since we actually received and acted on the messages but just dropped the ACK, so the server thinks we didn’t receive them. Then after about five seconds, we can look at both users to view the re-broadcasted message. While we should see duplicate messages, in reality, a client would not have received both messages because one would have been dropped, but in our scenario, we still receive the message but decide not to ACK it; it is worth noting that an ACK can be lost when sent from client to server resulting in this exact scenario. We use the same code as in Deliverable 3 for dropping ACKs.

Validation Results

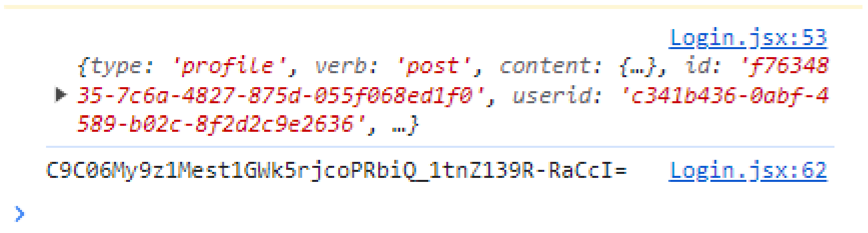
This section displays the results of the Validation Process.

1. Key Sharing

- Public Key Encryption Test

```
{
        "type": "profile",
        "verb": "post",
        "content":
        {
                "username": "Logan",
                "userid": "c341b436-0abf-4589-b02c-8f2d2c9e2636",
                "symmetric": "C9C06My9z1Mest1GWk5rjcoPRbiQ_1tnZ139R-RaCcI=",
                "encrypted_symmetric":
                "YRp1miduHYgLLmXRFOnMRaqqH70HdwNOb3pOsWtvBRjOaeXQ97c1Hq9Lj0
                Dntaou1xkvZK3olIP52AsFpIf2fbscgJFawl0AyDcbGixJKxE5XJzVMiRXeK5OBpU
                eQpnxIWde9q6XvIorx0AaHPAQMwog9sWoAxY4oTYxKjURDxAffknpfTmoEVB5
                +ujxYZ0xAFj1I+Tba99SYfSVJdd1nNwipqa1MuNR/xUcRm+mxaNQPKXvaxufutn
                Fc9/utInMjKSbi1W9NGv2eHKzCY057VZ4mHrmGBQvL6MWMWFcgod9UgNSvE
                m0R2hNeMgScDDxaZYonsd/FqCoq/nIL5NgIQ==",
                "status": true
        },
         "id": "f7634835-7c6a-4827-875d-055f068ed1f0",
        "userid": "c341b436-0abf-4589-b02c-8f2d2c9e2636",
        "ack_timestamp": 1700936617.600272,
        "useEncryptedSend": false
}
```

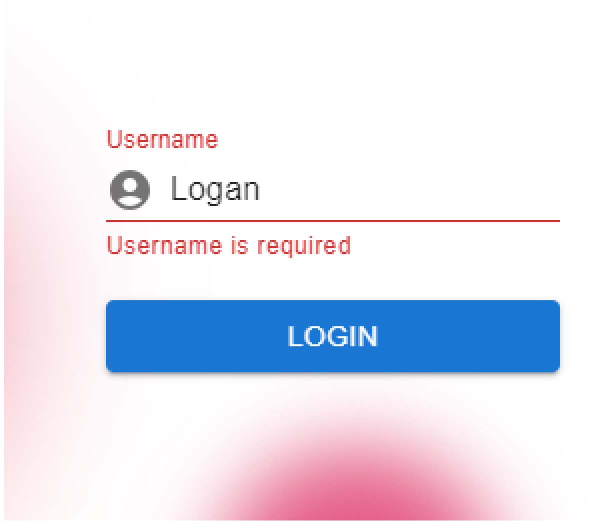
The above is the response from the server for creating a user whose username is “Logan”. We can see that the symmetric key was encrypted successfully. We also print out the decrypted result in the console:



```
                                                   Login.jsx:53
  {type: 'profile', verb: 'post', content: {…}, id: 'f76348
▶ 35-7c6a-4827-875d-055f068ed1f0', userid: 'c341b436-0abf-4
  589-b02c-8f2d2c9e2636', …}
C9C06My9z1Mest1GWk5rjcoPRbiQ_1tnZ139R-RaCcI=    Login.jsx:62
>
```

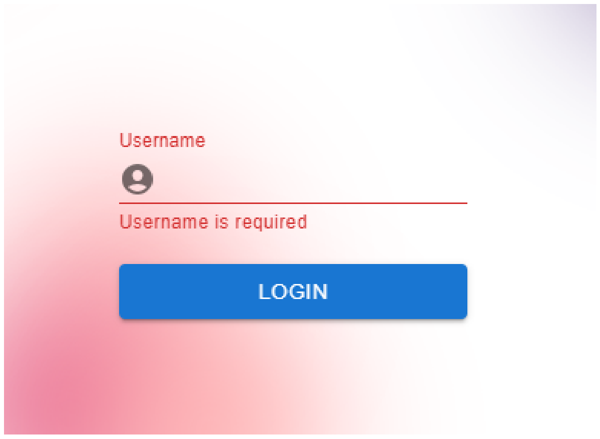
2. Profile Creation

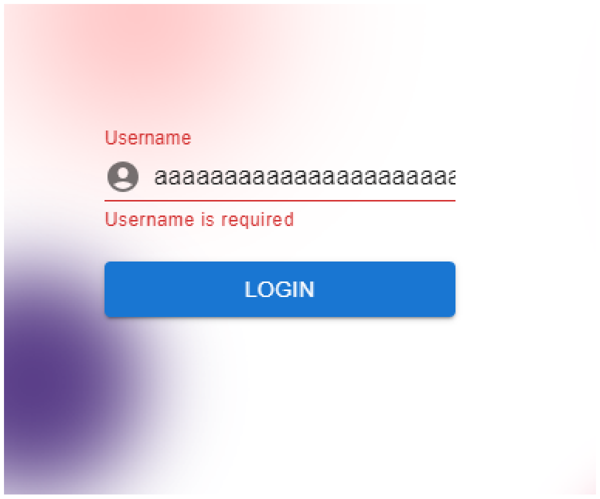
- Uniqueness Test



We see that “Logan” was not accepted. The error message for all failure cases is “Username is required”.

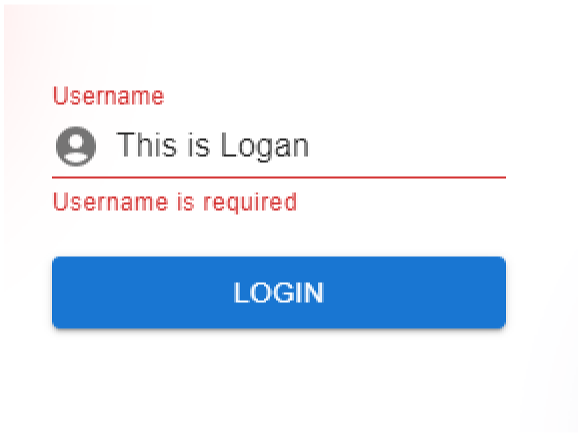
- Username Length Test





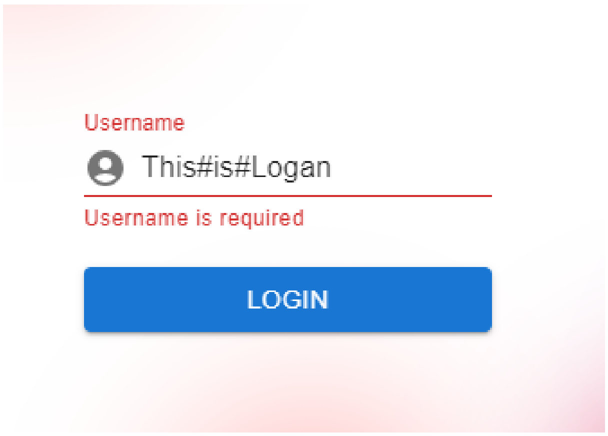
We see that “” was not accepted nor was “aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”.

- Alphabet Test


Username

This is Logan

Username is required


Username

This#is#Logan

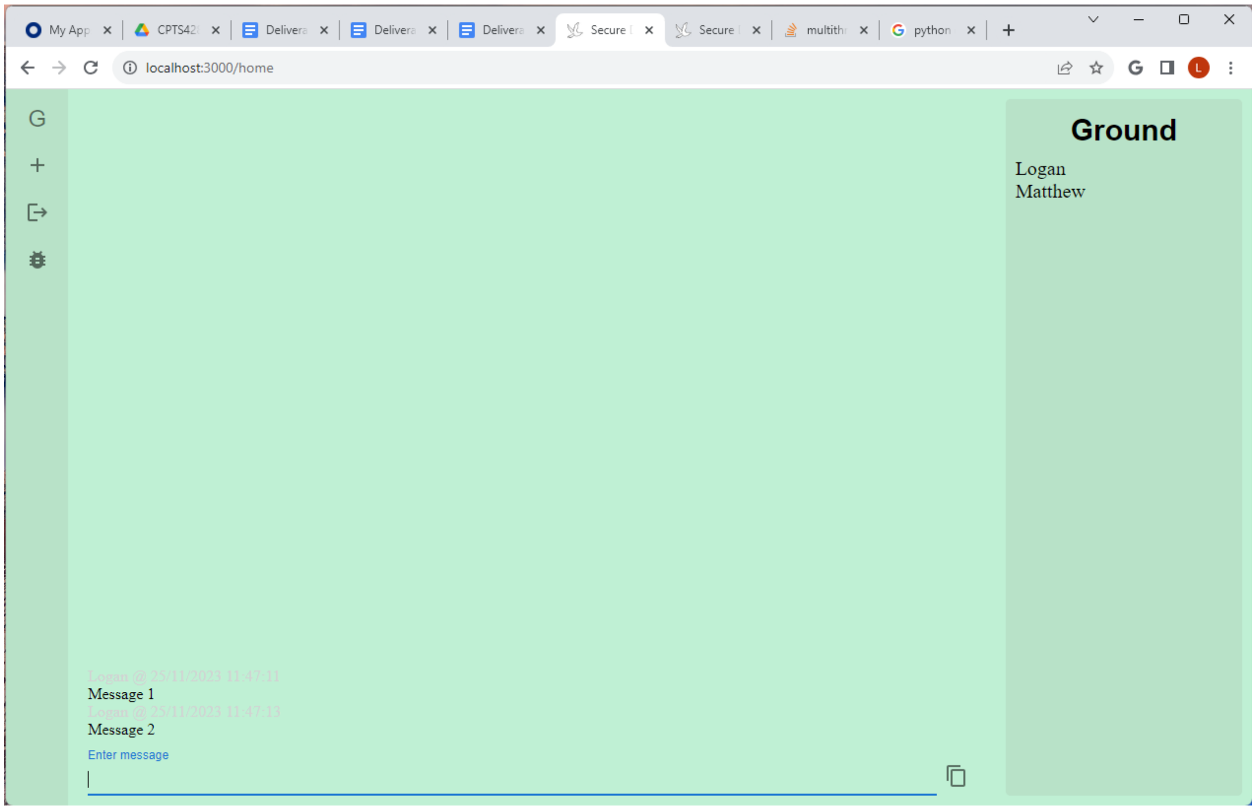
Username is required

We see that “This is Logan” and “This#is#Logan” were not accepted.

3. Message Resiliency

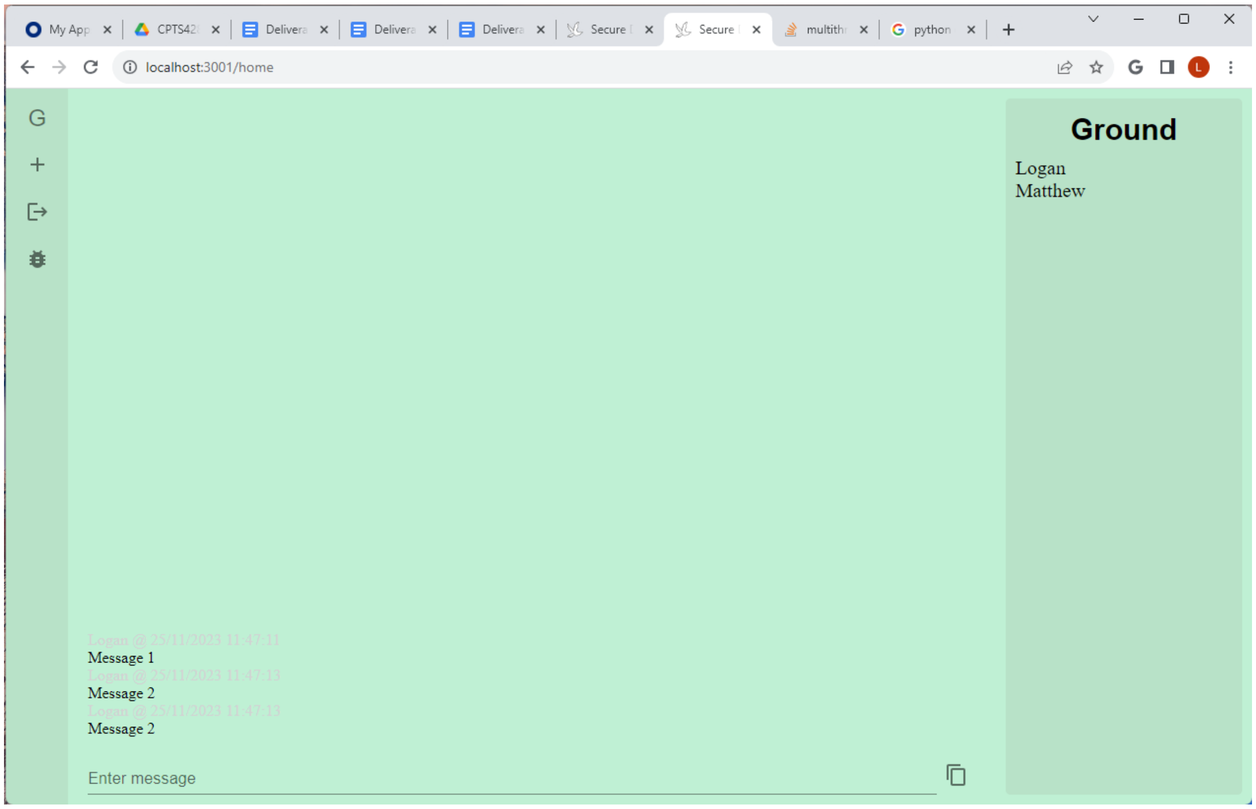
- 50% Drop-Rate Test

Logan Sends two messages. Depending on where we are currently at in the 50/50 ACK, the first or second message could be dropped.



Logan
Matthew

Logan @ 25/11/2023 11:47:11
Message 1
Logan @ 25/11/2023 11:47:13
Message 2



Ground

Logan
Matthew

Logan @ 25/11/2023 11:47:11
Message 1
Logan @ 25/11/2023 11:47:13
Message 2
Logan @ 25/11/2023 11:47:13
Message 2

Matthew waits for duplicate messages. We see that the second message was dropped because it is the one that is duplicated. We don't show it, but Logan also sees duplicate messages since his client drops 50% of ACKs and even though the message originated from Logan, Logan still receives messages the same way that Matthew does.

Conclusion

At the beginning of this document, we outlined the updated version of the software design, then discussed the validation process and outlined the validation results. The validation results tell us that our updated software design fixes the previous issues discovered in Deliverable 3. In the case of the symmetric key being visible as plain text, we now see that it has been successfully encrypted. Then in the case of abusing usernames, we were able to observe that uniqueness, length, and alphabet criteria were successfully enforced. Finally, in the case that a user does not receive a chat message, the server will continuously attempt to send the message to the user.

In addition to these security measures, our application continues to provide the same mechanisms seen throughout the other reports and expressed by our security goals. We use Fernet symmetric encryption to protect the content of messages shared between client and server. Fernet encryption is resilient to brute force, thus ensuring that the message is undecipherable while encrypted. A client is prevented from freely joining rooms that they don’t belong to due to each room’s id length and randomness. The audit log provides ACK success information and can be looked at offline to identify outages or other patterns.