Secure Messenger Design Report
CY 4740

We have implemented a server-client model where the server will be used for initial discovery and key exchange, and user authentication but messages will be transmitted between the clients (peer-to-peer). After obtaining a password-derived shared key with the server during login, the client will use that shared key to request information about another user from the server. This design currently supports 4 pre-registered users.

Login:

pre-registration (SRP):

We assume that all users have already been registered with the server and that the server already has a database that stores all the usernames and verifiers. A verifier is = g^w mod p where p is a large prime, g is a generator, and w is the user’s password. There is extra security provided because the password(W) is hashed before computing the verifier. We also assume p and g are constants known beforehand (received securely or pre-shared)
**note: In the real world, the server wouldn't have a file with the user’s passwords. The “creatingusers.py” is just to show how the g,w, and p values were filled out.

1. Client chooses a and sends g^a mod p

```
from connected user: {'type': 'SIGN-IN', 'username': 'aishu', 'g^amo
dp': 19576919665448427960350469273249626221869620995013050664902003
2861301102165995613767599684180179878280400916908052548829609032552
44641383566086106982628056765757735675201277522656157947667686133754
742360850560387181562883283195538711111241739868805819844640913055113
53041223434543971960286713450226816562372728321029125625572711965926
17164797765494543043008123464983648823586499674481158164087656101413
31529212232249573548378065732172974519319559103288036923366127283423
78420981121721414468472576758199996529874145118599996433905069591825
83651564199188637596969814927375791621278050443647653794624258503849
69385121835, 'port': 9901, 'ip': '127.0.0.1'}
```

2. Server chooses b, c_1, 32-bit number u
3. Server sends g^b+g^W mod p, u, c _1
4. Server computes shared key K = g ^(b(a+uW))mod p

5. Client receives g^b+g^W mod p, u, c _1

```
{"type": "SRP_RESPONSE", "g^b+g^W_mod_p": 136194357621114765069168618
340935152107378711608539593738438268942137735876699336877783707656702
773543432475803865979992590964915627608777259022612915608871528190368
677483959376131400248835176201425632387627239175830597422933687234739
872028983005848842616957680535133310955496124822085441610739504638334
890431929138163140607535697519563367587875819258819501054472266018568
172072264354397899129220553099464387568717297488801846404234345945172
801205234600548854380881306252089417851822568331461614900657032237748
088158628547502302887439055461852417571393253018633771270082327919464
45005323892027474989549067206979658736, "u": 3844360987, "c_1": 48314
267}
```

6. Client computes the shared key K = g ^(b(a+uW))mod p

- client hashes the password similar to the server before computing shared key

7. Client sends c_1 encrypted using the new shared secret key to prove to the server that it is the user and also sends c_2

```
from connected user: {'type': 'AUTH_MESSAGE', 'encrypted_c1': 'FzmdL
PqWGOWr+POrlII4aTDLFDJjWwNh8twWn/bNSZo=', 'c_2': 18856194}
```

8. Server decrypts c_1 using the shared key and verifies that this is the right user and also sends back c_2 encrypted using the shared key -> mutual authentication

```
{"type": "AUTH_RESPONSE", "encrypted_c2": "X4RfpB7VVaKsMyCybqbJHL4YID
qEFvsRtKl3afeRsQ=="}
Log in successful!
```

- uses AESGCM for encryption with the shared key

Tested scenarios

- Incorrect username/ Unregistered user

```
Please enter your username: netsec
Please enter your password:
User not found
Please enter your username:
```

- Incorrect password

```
Please enter your username: aishu
Please enter your password:
User verification failed
Please enter your username:
```

- Multiple incorrect passwords/ Timed out client tries to connect
  The user is temporarily locked after 4 incorrect passwords for 5 minutes

```
Please enter your username: aishu
Please enter your password:
User verification failed
Please enter your username: aishu
Please enter your password:
User verification failed
Please enter your username: aishu
Please enter your password:
User verification failed
Please enter your username: aishu
Please enter your password:
User temporarily locked out. Try again later.
aishuvinod@Aishwaryas-MacBook-Air-2 client %
./chat_client.py
Please enter your username: aishu
Please enter your password:
User temporarily locked out. Try again later.
```

- The same user cannot be signed in on different client terminals

```
aishuvinod@Aishwaryas-MacBook-Air-2 client %
./chat_client.py
Please enter your username: mallory
Please enter your password:
User already logged in
```

```
aishuvinod@Aishwaryas-MacBook-Air-2 client %
./chat_client.py
Please enter your username: mallory
Please enter your password:
Log in successful!
```

- Server is offline when client tries to login

```
aishuvinod@Aishwaryas-MacBook-Air-2 client %
./chat_client.py
Please enter your username: aishu
Please enter your password:
No data received from the server. Exiting.
aishuvinod@Aishwaryas-MacBook-Air-2 client %
```

```
aishuvinod@Aishwaryas-MacBook-Air-2 server
%
```

- Server goes offline in the middle of a connection

```
aishuvinod@Aishwaryas-MacBook-Air-2 client %
./chat_client.py
Please enter your username: aishu
Please enter your password:
Log in successful!
Please enter command:
Server is shutting down. Goodbye!

Exiting the client.
aishuvinod@Aishwaryas-MacBook-Air-2 client %
```


```
8593869634565195129564934841851215178995656
0436816837376215352246941236311166425610547
1031116713161449528948523814277782255955863
8241822027348492958560356948024945040466826
1286880900666990209632844912382497193035213
7535552140148721804751871514402814400679553
2224167909743239063368146285388519636816064
5448976454892099223058332115884907331033440
8162416071873248130253336405849918710339028
4251472969193607509413546348260374168827133
0730352923790732988137476540666078855504681
9686939330006173989848125579999946307058840
7742107359888203370374804483787240670751663
192737033542, 'port': 9090, 'ip': '127.0.0.
1'}
from connected user: {'type': 'SIGN-IN', 'u
sername': 'aishu', 'g^amodp': 591851197619З
4470917626586248504352966578760785938696345
6519512956493484185121517899565604368168373
7621535224694123631116642561054710311167131
6144952894852381427778225595586382418220273
4849295856035694802494504046682612868809006
6699020963284491238249719303521375355521401
4872180475187151440281440067955322241679097
4323906336814628538851963681606454489764548
9209922305833211588490733103344081624160718
7324813025333640584991871033902842514729691
9360750941354634826037416882713307303529237
9073298813747654066607885550468196869393300
0617398984812557999994630705884077421073598
8820337037480448378724067075166319273703354
2, 'port': 9090, 'ip': '127.0.0.1'}
Data from ('127.0.0.1', 59583): {'type': 'A
UTH_MESSAGE', 'encrypted_c1': 'l7r4J70oUu3/
240BhrfIreBj+If7IhH3HCkPH4eZikg=', 'c_2': 8
9331470}
from connected user: {'type': 'AUTH_MESSAGE
', 'encrypted_c1': 'l7r4J70oUu3/240BhrfIreB
j+If7IhH3HCkPH4eZikg=', 'c_2': 89331470}
^C
Server shutting down...
```

List Command

```
Please enter command:  list

<- Signed In Users: alice, mallory, aishu, bob
```

tested scenarios:

1. Someone logs out/exits: the user is removed from the list
2. User is not added to the list if they fail password verification or are unregistered
3. Since users cannot be logged in on multiple terminals, they also cannot be duplicated in the list output

---

Messaging (Expanded Needham-Schroeder inspired): (Send command)

1. Client A sends a message to the server after they have logged in which is encrypted using the shared key K with the server which was created during login. This message will contain the client's username, the username of the client they want to message, and a nonce
```
# Create the message dictionary
message_dict = {
    'from': user,
    'to': username,
    'nonce_1': nonce_1
}
```
2. The server will decrypt that message coming from the client
3. The server will send back another message encrypted with their shared key which contains another nonce, a shared key that the two clients can use for initial communication, and the IP address of client B so that client A can message them, This message will also contain another message encrypted using the shared key between the server and client B containing the client to client shared key

```
# Encrypt the new shared key with the shared key between the server and the from user
message_content_to_A =  {
    "nonce_1": nonce_1,
    "shared_key": shared_key,
    "to_address": recipient_address,
    "ticket_to_B": ticket_to_B
}
```

```
ticket_to_B_contents = {
    "shared_key": shared_key,
    "from_user": from_user,
    "sender_address": sender_address,
}
```

4. Client A decrypts the part of the message that is encrypted with its shared key with the server and retrieves the shared key with the user it wants to talk to and their address
5. Client A will send the message encrypted with the shared key between client B and the server to client B (ticket to B) which contains the shared key between the two (client A cannot decrypt this message). The client will also send its identity and a Nonce.
6. Client B will decrypt the client-client shared key using the key it has shared with the server.
7. Using the retrieved shared key, client B will decrypt the Nonce and send back the Nonce - 1 along with its nonce
8. Client A verifies that it can decrypt the message and that the Nonce sent is the nonce that it had sent (- 1)
9. Client A sends back Client B's nonce - 1
10. Once Client B verifies that the nonce sent by Client A is the nonce that it had sent - 1, Mutual authentication is achieved.
11. Future messages are encrypted using this shared key until the session ends (the process is repeated)

Tested scenarios:

1. Multiple users talking to each other

```
aishuvinod@Aishwaryas-MacBook-Air-2 clien
t % ./chat_client.py
Please enter your username: aishu
Please enter your password:
Log in successful!
Please enter command:  send mallory hi
Please enter command:
```

```
aishuvinod@Aishwaryas-MacBook-Air-2 clien
t % ./chat_client.py
Please enter your username: mallory
Please enter your password:
Log in successful!
Please enter command:
<- From aishu: hi
Please enter command:  send bob hello
Please enter command:
```

```
aishuvinod@Aishwaryas-MacBook-Air-2 clien
t % ./chat_client.py
Please enter your username: alice
Please enter your password:
Log in successful!
Please enter command:
<- From bob: whats up
Please enter command:  send bob what do u
 want
```

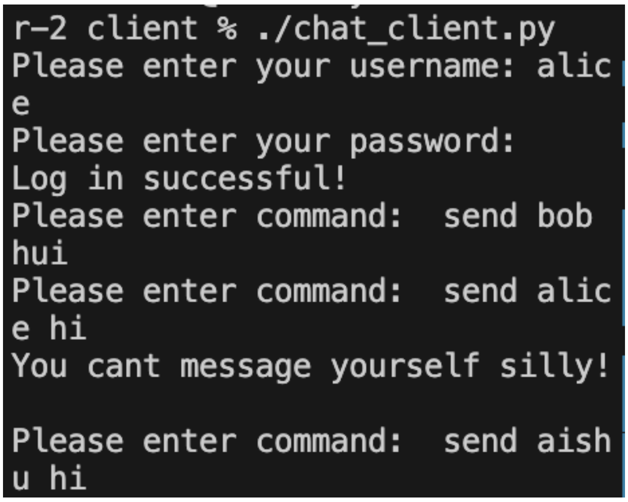
```
aishuvinod@Aishwaryas-MacBook-Air-2 clie
nt % ./chat_client.py
Please enter your username: bob
Please enter your password:
Log in successful!
Please enter command:
<- From mallory: hello
Please enter command:  send alice whats
up
```

2. Shared keys are forgotten if one of the users exits, or if the session ends between the clients (and server as well)
3. When a user tries to message another user is not logged in or doesn't exist

```
Please enter command:  send aishu hi

<- User offline or cannot be reached. Try
 again later.
Please enter command:
```

4. When a user tries to message themselves



```
r-2 client % ./chat_client.py
Please enter your username: alic
e
Please enter your password:
Log in successful!
Please enter command:  send bob
hui
Please enter command:  send alic
e hi
You cant message yourself silly!

Please enter command:  send aish
u hi
```

Features

Online and offline dictionary attacks:

- Rate Limiting: We have implemented rate limiting on login attempts. After 4 failed password attempts, the server will temporarily lock the user account which will prevent attackers from repeatedly trying different passwords within a short period.
- Users are required to set complex passwords containing a combination of uppercase and lowercase letters, numbers, and special characters which will make it more difficult for attackers to guess passwords through brute force
- SRP: When a user registers with the server, instead of storing the password directly, the server stores a verifier v, which is derived from the user's password. So, even if the server database is compromised, an offline dictionary attack is not possible. If a random salt s is used during user registration, the salt combined with the password to store provides protection. This ensures that even if two users have the same password, their verifiers will be different due to the unique salts. This prevents attackers from precomputing a dictionary of hashes/rainbow table for common passwords since they would need to compute hashes for each password with every possible salt
- We have used a secure and computationally expensive hash function SHA-256 for password hashing.

Denial-of-service attacks

- The same user cannot be connected to multiple terminals. This way, once a user has connected via the client they cannot connect from another client until the original client has

disconnected. Meaning with our 4 users at most 4 active client sessions can be active
- If a user is trying to brute force a password they get timed out after 3 incorrect attempts for 5 minutes
- New users cannot register on their own
- Unknown users will not be able to attempt to connect

End-point hiding -

While we don't have this feature implemented currently, we would implement it in the following manner.

- Clients are pre-configured with the server's public key and they encrypt their initial message with that key. This way, the client's identity is protected
- The client generates a random public key and sends it to the server in the initial message
- The server encrypts the message going back with the client's public key. This way, no one knows who the endpoint user is.
- Past that point, everything is encrypted using the shared key.

Perfect forward secrecy - The shared session key between the client and server is forgotten after every session and the shared key between any two clients is also forgotten and regenerated every time. This way, past communications cannot be decrypted

Other special features added

1. The getpass.getpass() function reads the input from the user without echoing it back to the terminal, providing a more secure way to input passwords. Although you don't see the characters as you type them, the password is still being captured and processed by the script without being displayed on the screen.
2. Admin password for starting the server
3. The server password is obfuscated
4. Thorough error handling

Design Vulnerabilities

1. The server right now can decrypt all the messages between clients since the client uses the shared key provided by the server to communicate. If the server was a malicious actor this could be dangerous and in general, the server doesn't have to be able to decrypt client-client messages. How we would implement it:
    - Once the shared key is provided by the server and the clients mutually authenticate each other, perform a diffie-hellman between the clients to create

a new shared key that only the clients know. This new key will also be forgotten once the session ends.

2. Lack of endpoint hiding