Secure Programming

Semester 2, 2025

Advanced Secure Protocol Design

Reflective commentary, Group 43

THE UNIVERSITY of ADELAIDE

Produced by:
Mohammad Waezi a1853470
Mikaela Somers a1852586
Zi Tao a1916843
Pahlavonjon Odilov a1827303

Table of Contents

1. Introduction

A crucial component in modern software engineering is safe and secure programming, through the ability to identify and mitigate potential vulnerabilities. This assignment goal is to create a secure, multi party chat system capable of connecting with other students, to share private and public messages, transfer files and list all current members.

To begin, the design required the creation of a collaborative, standardised class wide protocol, known as the Secure Overlay Chat Protocol. This was then followed by group implementation of the agreed upon functions and methods, with a focus on strong encryption and security measures. Within this implementation, several intentional backdoors and vulnerabilities were added to a ‘backdoored version’, which other students were tasked to find through code review and an ethical hackathon competition. Feedback provided by peers, such as strengths and weaknesses of the code functionality and its security, were taken into account and reflected upon accordingly.

2. SOCP protocol reflection

The Secure Overlay Chat Protocol (SOCP) is a decentralized communication framework designed to ensure secure and interoperable messaging between distributed servers and clients. Its architecture emphasizes end-to-end encryption (E2EE) and trust decentralization, achieved through the use of RSA-4096 asymmetric encryption.

Asymmetric encryption is used as it allows two parties to exchange encrypted messages without sharing a secret key directly. Each user has a public key (used for encryption) and a private key (used for decryption), eliminating the need for centralized key management and reducing the risk of compromise. The choice of 4096-bit RSA keys instead of the more common 2048-bit keys offers stronger cryptographic resistance against brute-force attacks, providing long-term security, suitable for distributed systems where key rotation may be infrequent.

In SOCP each user connects to only one local server and all servers form a mesh network, where messages are routed based on user mappings. This design removes dependency on a central authority, avoiding a single point of failure. The resulting complexity in synchronization and message routing is mitigated through gossip-based updates and unique message identifiers. Servers regularly exchange presence and routing information, ensuring that user states remain consistent across the network while preventing message duplication or loops.

Version 1.3’s shift to RSA-only encryption simplifies the cryptographic model but introduces performance trade-offs. While RSA provides robust security, it is computationally expensive and unsuitable for large data transfers. Previously, AES (a symmetric cipher) was used alongside RSA for encrypting message payloads more efficiently. In a decentralized chat system like SOCP, a hybrid encryption model using RSA for key exchange and AES for bulk data encryption would likely achieve a better balance between speed and security.

SOCP's dual digital signature structure ensures message integrity and non-repudiation, but its inclusion of "backdoors" highlights how even small implementation flaws and other vulnerabilities can be brutally exploited and compromise security. Overall SOCP demonstrates the balance between theoretical cryptographic strength, performance and practical system design in secure decentralized communication.

3. Implementation reflection

3.1 Overall design

As the assignment mentions, we had to follow an agreed upon protocol created by all students. Once the protocol was finalised and published as SOCP v1.3, we officially began the development process. Our implementation strictly followed the SOCP v1.3 specification, and the system was fully built in Python. We divided the code into multiple files such as *server*, *client* and the *introducer* for modularity. To begin, we started off with setting up a functional WebSocket connection on one device using two different terminals to simulate a client and a server. We then proceeded with establishing the connection across multiple devices and made sure it could send and receive messages. That became the foundation of our system which we built upon by adding new features one at a time.

3.2 Assumptions

At the start, we assumed our program would run the same way on any device if it had an IP address. Later, we found that wasn't the case when running the server on WSL. The Linux subsystem uses its own internal IP that isn't visible to the LAN, so clients couldn't connect even though it worked perfectly when hosted directly on Windows. We also assumed that every group would fully follow the SOCP v1.3 standard, but small differences in things like JSON formatting or RSA setup caused signature verification to fail. That's when we realised even a tiny change, like key order in JSON, could break communication between servers. Another assumption we made was about the "Introducer" part in the SOCP spec. It was written vaguely, so we built our own introducer server that simply connected new peers to others. That design worked well and matched the protocol's intention. These lessons helped us understand how easily assumptions can break interoperability if not tested properly.

3.3 Code quality:

The codebase follows a modular architecture. According to the SOCP v1.3 specification, we implemented a clear separation between client-side logic, server-side logic, and cryptographic functionalities. Besides, each Python file follows the PEP 8 style guide and includes comprehensive docstrings for all modules and functions, which ensures readability and maintainability. We also implemented exception handling for every necessary scenario, which improves traceability and makes debugging processes more efficient. Further details are provided in the following section, "Error Handling". Additionally, security checks such as RSA signature verification, duplicate user detection, and user authentication verification are strictly

enforced, preventing malformed messages or unauthorized users. Overall, the implementation demonstrates strong reliability, security, and robustness.

3.4 Error handling:

Our application implements structured error handling following the SOCP v1.3 specification. First, we implemented comprehensive validation and clear error responses during the user authentication process. During registration, the server checks for duplicate usernames and invalid payloads, returning error codes such as `NAME_IN_USE` or `UNKNOWN_TYPE` when issues are detected. In the login process, if the username does not exist, the server sends `USER_NOT_FOUND`. If the client fails to respond to the challenge correctly or the HMAC proof does not match, it returns `INVALID_SIG` or `BAD_PASSWORD`. Each error triggers a structured ERROR frame to the client, ensuring that failures are communicated safely and effectively.

In the process of message delivery and inter-server communication, we also implemented corresponding error handling for each case. Once a standard envelope frame is sent, the payload signature is verified. If the verification fails, the exception is caught and an error message such as “Invalid signature on [specific frame name]” is printed. This approach ensures that signature or transmission errors are captured and reported efficiently.

Additionally, the system gracefully handles node failures by cleaning up connections and notifying other peers. For example, when a client disconnects, the server logs “User {username} disconnected”, and all other users automatically update their user lists accordingly.

3.5 Testing and interoperability:

Testing was conducted at both the module and system levels. Unit tests were used to verify core components such as encryption and decryption, signature verification, routing logic, and error handling. Integration tests were performed for both inter-server communication and communication within a single server, demonstrating that the system can exchange messages over WebSocket with correct forwarding and presence gossip in both directions. Additional integration tests were conducted across multiple devices to ensure that IP addresses were correctly recognised and connections established successfully. We successfully exchanged `/tell`, `/all`, and `/file` messages, confirming correct routing. The `/list` command also functioned reliably in all test scenarios.

We also learnt a lesson from the testing process. During both the development and code review processes, we discovered a common issue that appeared in many groups’ implementations, including ours. The issue concerns the order in which servers and clients are launched when testing chat functionalities across different servers. For example, if we launch two servers first and then start one client under each server, the chat between the two clients works normally. However, if we launch one server and its client first, and then start the second server followed by its client, one of the clients will not recognise the other. As a result, a `USER_NOT_FOUND` error occurs. This problem highlights the importance of thorough and systematic testing for complex distributed systems like this project. Only if we carefully test different connection and

startup orders, the system behaves correctly under all possible scenarios can be ensured. Additionally, testing was completed with one other group to ensure the system functioned as expected before submission.

A detailed test plan and corresponding test logs are available in our GitHub repository and included in the appendix 2 of this report. We also provide a detailed demonstration of how our code runs by chatting with our own implementation in the appendix 2 of this report.

4. Backdoors and ethical vulnerabilities

Several ethical backdoors and intentional vulnerabilities were implemented within the backdoored submission of our project, ranging from encryption weaknesses to log in exploitations. Please see appendix 3 for a detailed PoC of each.

To begin, we implemented a dual log in feature which allows several users to sign up using the same username. The section of code was removed which checks for existing usernames and prevents impersonation, allowing any user to log in with the same username as another logged in user. Additionally, if the username already exists, the system will not announce a new user has joined, as not to alert users of the duplicate log in. When the ‘/list’ function is typed, the duplicated username will only appear once. All users with the same username will receive messages sent to the username, and all users are capable of sending messages from the username. This allows a malicious user to impersonate any other user by creating a new account with an already existing username, enabling them to receive all messages intended for the original user, without the original user or any other user knowing.

Secondly, a master password has been hardcoded which allows log in access to any user. This password is found as the variable ‘ FOLDER_DIRECTORY = ‘data’ ’ as not to create an obvious backdoor. A backup key has been created which is accepted by the server and authenticated just like a normal log in. The HMAC code snippet can be seen in the appendix. This backdoor allows any user to log in with either their correct password or the master password, ‘data’, and similarly, an attacker can then gain access to any user’s account using this bypass.

Next, a hashing vulnerability was inserted where a predictable hashing scheme is used with non random salt. The salt is generated using a user’s username padded with null bytes to the length of 16 bytes, which is easily predictable. The function which generates this salt can easily be found within the code, and hence an attacker could easily calculate a user’s salt from their password. A detailed PoC of how this could be used by an attacker to gain access to a user from any client application can be found in the appendix.

The system additionally has vulnerable and weak key acceptance, through false assurance that it only accepts 4096 RSA keys. However, the code has been edited to allow acceptance of 1024 keys, as seen in the appendix. This allows an attacker to create an intentionally weak key, which will be accepted and distributed across the network. While this is not a direct backdoor, it is a

vulnerability which decreases an attacker's difficulty in decrypting confidential data regarding internal communications of the network.

Another vulnerability inserted into the code relates to the incorrect authentication of public keys. The network does not prove that the public key belongs to the claimed user, meaning an attacker can force the sender of a message to unknowingly encrypt using the attacker's key. This can then be decrypted by the attacker and sent onwards to the intended recipient. This approach requires the attacker to inject code into certain parts of their own server, seen in the appendix, which creates a spoofed envelope telling the receiver that the attacker's key is actually a user's key.

Finally, when a user sends a file to another user, it is automatically downloaded onto the recipient's machine, without the ability to confirm or deny the incoming file. This could allow an attacker to do several things; they may send a file to a user intentionally containing malware, or several very large files aiming to fill disk space. Additionally, through manipulation of the file name itself, an attacker is able to change the location a file is downloaded to, with the potential to move up directory levels, such as traversing to startup folders which may run the file when the user's machine starts up.

These six backdoors range from intentional log in and password vulnerabilities to weaknesses within encryption methods, and message and file sending vulnerabilities, demonstrating a deep understanding of a range of exploitation methods and secure programming practices.

5. Feedback received

Our group received peer review feedback from 12 students, referenced in appendix 5. This feedback covered a range of strengths, weaknesses and recommendations, as well as identifying several of our deliberate backdoors. 8/12 reviews identified the master password, 'data', as an alternate log in method, while most discovered the weak hashing scheme. These backdoors were simple to find through a thorough code review, while other backdoors, such as the transport verification and acceptance vulnerabilities, were more difficult to find and hence only 4 groups identified this as a security risk. Surprisingly, the weak key acceptance vulnerability was found by only a few students despite being a simple to implement and identify backdoor. A handful of students discovered flaws with file transfer, particularly DoS from large file sizes, though did not consider destination traversal. Many identified the security flaw in plaintext storage of the keys, as expected, and two students discovered the ability to have several duplicated user name log-ins. Overall, each intentional backdoor was discovered by at least one student, though only the obvious vulnerabilities such as hardcoded bypasses were discovered by the majority. The reviews additionally discovered slight issues and suggested improvements that we otherwise did not consider. One example was the ability to choose any user to attempt to log in as, not just the username previously created / used. Currently, a user creates an account and then is only given the option to log in as that user, not any other. This was done to prevent users attempting to log in as other users, although this feedback was

helpful in considering that a user may want to have several accounts. Additionally, it was suggested that we replace silent exception handlers with explicit logging and fail-closed behaviour, as well as removing plaintext storage and viewing of keys. Another helpful suggestion was limiting the file size that a user can receive to prevent denial of service attacks. The received feedback and suggestions given were extremely helpful and provided otherwise missed functionality issues. Having several students not involved in the development process review the code in such a detailed manner provided a fresh perspective on the security, functionality and user friendliness of the system.

6. Feedback given

Each group member reviewed three other groups' implementations which resulted in a total of 12 reviews. Detailed reviews are attached in the appendices.
Our review process mainly included manual testing of each code file, as well as automated testing with tools such as *'Bandit'.* We used VM environments such as *'Oracle Virtual Box'* as instructed to maintain a safe and isolated setup. We focused on finding vulnerabilities, cryptography weaknesses and most importantly protocol compliance with the SOCP v1.3.
We began each review by following the group's README file and establishing a server and clients to confirm functionality. Some implementations required extra efforts to get running. For example group 32's code could not connect a client to a server. We handled such situations by contacting the respective group members for assistance.
Once the systems were running, we tested features such as /tell, /all and transferring files through /file for every group.
We ensured our feedback was constructive and detailed. We found most vulnerabilities through manual inspections. The most common ones were weak or missing authentication checks. Many groups allowed users to login without verifying their identity or private key and some stored private keys in plain text. We provided clear recommendations such as enforcing proper key validation, encrypting stored secrets, and verifying message signatures before processing. Overall the process helped us better understand secure programming and how easily validation mistakes can lead to major vulnerabilities.

7. Use of AI

AI played a big role throughout our project. We used it to help write code that was free of syntax and logic errors and followed the SOCP v1.3 rules. We constantly reminded it about the protocol and assignment requirements, so it stayed consistent with what we needed.

We asked AI to explain and teach us new concepts we hadn't learned before, like RSA encryption and WebSocket handling. That really saved us time because we only needed to learn things to the extent required to build and handle the implementation. It also helped us fix bugs when something broke, and assisted in the implementation of ethical vulnerabilities and

backdoors, particularly in finding bugs that prevented the backdoors from being functional and breaking down how they could be exploited. One main weakness of AI was that sometimes it would add a new feature without considering the existing ones, which ended up breaking older parts of the code. In those cases, we had to manually rebuild or adjust the feature to make it work with the rest of the system.

Even though AI helped us significantly in writing and cleaning our code, we still had to handle all the manual setup and configuration such as IP addresses across different devices and test everything ourselves to ensure they worked together. Using AI made the process faster and taught us a lot, but we still had to think through the logic, adjust the structure, and solve problems on our own to get the system running as desired.

8. Course material / past experience reflection

This project required knowledge from a number of past courses and previous projects, combining techniques in web development, encryption, algorithm development and distributed systems architecture to implement a secure multi party chat system. Particularly, knowledge gained from courses such as Computer Systems were crucial in understanding architecture, memory, compilers and languages, while algorithm based subjects such as Algorithm Design & Data Structures, and Algorithm & Data Structure Analysis provided skills crucial for developing and handling logic for user presence, message delivery and error handling within the chat system. Skills from previous subjects created such as these were crucial in our ability to develop the chat system.

The required completion of RangeForce modules within this course provided a hands-on learning experience in secure programming techniques, and clearly demonstrated the consequences of insecure coding practices. This was achieved through the use of software such as Cutter and OWSAP ZAP, where static and dynamic testing was used to examine several malicious and non malicious files. In combination with theory from lectures, this experience provided us with the knowledge and skills to minimise vulnerabilities within our code, and insert intentional backdoors. The project completion deepened our understanding of the importance of secure programming practices and contributed significantly to our cybersecurity skillset, which will play an important role in future university and industry projects.

9. Contributions of group members

In the SOCT project the team of four collaborated effectively with unique roles for each member. Two members focused primarily on core development and implementing key features such as peer-to-peer communication, message encryption, and distributed data handling. They also ensured integration between the client and server components. The other two members contributed by conducting testing and debugging, identifying performance issues and verifying system reliability under various network conditions. The testing feedback played a vital role in

improving the final product's stability and usability. One member focused primarily on introducing backdoors and vulnerabilities in the code and developing a proof of concept for these additions.

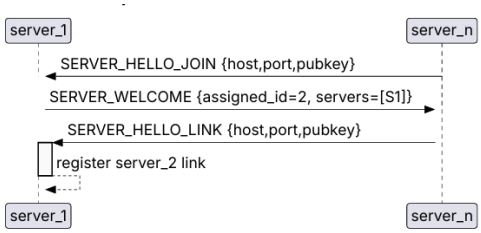
10. Conclusion

The SOCP project provided invaluable experience in designing and implementing a secure, decentralised chat system. It underscored the importance of end-to-end encryption, decentralised trust and strict protocol compliance. Introducing intentional backdoors offered a unique opportunity to understand how vulnerabilities are detected and exploited, enhancing our knowledge of ethical hacking and vulnerability management. By implementing robust security features and addressing identified vulnerabilities, we developed a resilient decentralized chat system. Rigorous testing and peer reviews helped identify and resolve remaining bugs, highlighting the critical role of systematic validation in software development. Overall this project strengthened our understanding of secure programming, distributed systems and ethical vulnerability management. This provides a strong foundation for future endeavors in software development and cybersecurity, particularly connecting to the current global demand for safe programming practices within software engineering and reinforcing the importance of secure programming in industry.

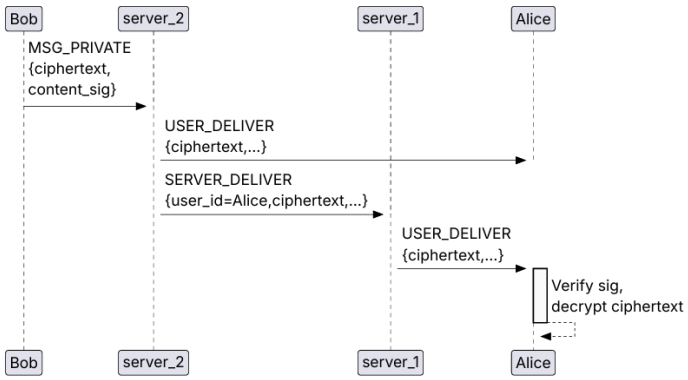
Appendices

1. Protocol Design Document

We followed the SOCP v1.3 for our implementation. The diagram below summarises the key elements:



17.2. Direct Message (Bob → Alice)



2. Testing and running

Proving the tests on github README have passed:

Test Plan & Current Status

ID	Role	Action	Pass Criteria	Fail Criteria	Current Status
1	New user	Sign up by typing 'signup' after running `python client.py`	The system will ask the user to enter a username and a password.	The system runs into error.	Pass
2	New user	Sign up with a duplicate username with other users.	The system will give a `NAME_IN_USE` error.	The user can sign up successfully.	Pass
3	User	Log in to the system by entering a correct password.	The user can successfully log in.	The system runs into error.	Pass
4	User	Log in to the system by entering a wrong password.	The system will give a warning about 'incorrect password'.	The user can successfully log in.	Pass
5	User	Typing `/list` command.	All the online users (no matter they are in the same server with the user or not) will be listed in the user's terminal.	The system runs into error or the user list is not complete.	Pass
6	User	Typing `/all {message}` command.	The system will show a message about 'the message has been sent and the message will be sent to all online users.	The system runs into error or the message has not been sent.	Pass
7	User	Receiving `/all {message}` command from others.	All the online users (no matter they are in the same server with the sender or not) will receive this message.	The system runs into error or the message is not delivered.	Pass

8	User	Typing `/tell {username} {message}` command.	The system will show a message about 'the message has been sent and the message will be sent to the specific user.	The system runs into error or the message has not been sent.	Pass
9	User	Receiving `/tell {username} {message}` command from others.	The user will receive the message from the sender.	The system runs into error or the message is not delivered.	Pass
10	User	Typing `/file {username} {file path}` command.	The system will show a message about 'the file has been transferred and the file will be sent to the specific user.	The system runs into error or the file has not been transferred.	Pass
11	User	Receiving `/file {username} {file path}` command from others.	The user will receive the file from the sender, and the file will be in the `downloads` folder.	The system runs into error or the file is not delivered.	Pass
12	User	Try to send message or file to a user who does not exist.	The system will give a `USER_NOT_FOUND` error.	The system runs into other error.	Pass
13	User	Disconnect from the server.	The server will print 'User {username} disconnected', and every other users will receive the notification and remove the user from the list as well.	The disconnected user is still in other users' online user list.	Pass
14	Introducer	Disconnected.	All servers will get a notification and they will be discounnected.	Servers are still connected.	Pass
15	Server	A new server or a new client joins the network.	No matter in what order that a server or client joins the network, the new client will get to know all other clients' infomation and others will get to know the new client's information though `USER_ADVERTISE`.	In some cases, some clients will get `USER_NOT_FOUND` error due to incorrect `USER_ADVERTISE`.	Pass

Test 1:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? signup
Choose a username: mj
Choose a password:
✅ Local keys created for mj (3075c216-3d80-410a-9827-61ce95dc85c9)
📮 Registering with server…
🎉 Registered on server. Your user_id is 880be23e-4afa-4238-963c-0999f2fbaa1e

moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$
```

Test 2:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? signup
Choose a username: mj
Choose a password:
✅ Local keys created for mj (b79e4ae3-449f-4ea3-a13c-af2d0a787238)
📮 Registering with server…
❌ Registration failed: {'type': 'ERROR', 'from': 'server', 'to': '*', 'ts': 1761459919942, 'payload': {'code': 'NAME_IN_USE', 'detail': "Username 'mj' exists"}, 's
ig': 'PLx56OmmY670gML7CHinEwVW0aUAsAnupvKEgJ9xGiZp0ZeiuiVVBA3QZ_pwPAU0Mj6wnDhbbfCaIfcwsXlCWBnbFZJ_Jb4r1EYRI9bsLfrtm95Vzu1_kpiHnTdpWIpUg_j36Xzx5bpj40A1qJ5lU67xzTpWxF
H567p9ZATGS2lWD2nOrmJhCBe9MXmVEFj1hpAix6tkndMqnmk1pLaCHMoJpjZqftu-5_hani909rujPBGanOpV--5sFT5x4-lxOZlbgGIF5BpUCgmsA8Bw7HHQ7_H5tBR6NWxYSCxIRqLoHwXbD6_9G0fgIOY6XMIF1n
ciUaaQ4BtAqtRc1Y36Ru77LbvQCcyJ2qOtFye5bYLcy6_iZI9O3S17FQD94WRXYxFc5-knw7mrQgRIlpBHnK4YJ5HM2DWRnxPqAM62L_i_YYF3x6SGOU376ULFhkf8iTfJPqC3iasNuwtJswHAHmoxg7aFCibQL48PFO
sx7QxxH5mzRIhQgdk-1Vdx_gkTFAQhFbvDZDUDFq6VS2JBr_FZewT5edK7zl0Ir-9eqvu70ezGvIopOJ0nloa7rRMprYzynpI217TM6K85PXq83f3Wad8L-8OhyqckEE2fsDq7ektvsL_hoYzPov6yU2Cfe26KyTc2IE
09AM2qA4cfdHvcroKnWLezaf1vWp3d9_Y'}

moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$
```

Test 3:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? login
Password for mk:
🔌 Connected to server at ws://10.13.106.56:9001
>
```

Test 4:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? login
Password for mk:
⚠️Incorrect password, please try again.

moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$
```

Test 5:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? login
Password for pp:
🔌 Connected to server at ws://10.13.106.56:9001
> 🔑 Learned pubkey for mk (9617205b…)
/list
Known users: mk
>
```

Test 6:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? login
Password for pp:
 Connected to server at ws://10.13.106.56:9001
> Learned pubkey for mk (9617205b…)
/list
Known users: mk
> /all hi
/all sent to 1 users (self excluded)
>
```

Test 7:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? login
Password for mk:
 Connected to server at ws://10.13.106.56:9001
> /list
Known users: (none)
> Learned pubkey for pp (b6bf22ae…)

Public from pp: hi
```

Test 8:

```
Public from pp: hi
/tell pp helloo
Sent DM to pp: helloo
>
```

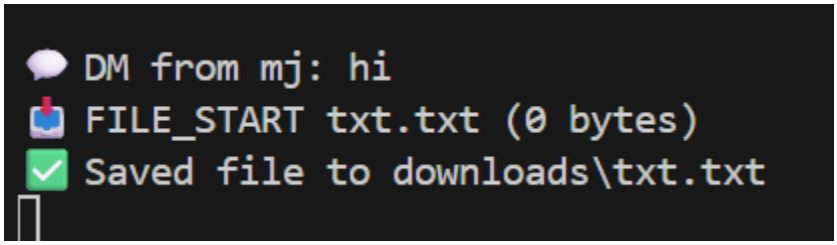
Test 9:

```
moham@Mohammad MINGW64 ~/OneDrive/Desktop/SEM2_2025/Secure Programming/SECURE_PROJ/Decentralised-chat/clean_version (main)
$ python client.py
signup or login? login
Password for pp:
 Connected to server at ws://10.13.106.56:9001
> Learned pubkey for mk (9617205b…)
/list
Known users: mk
> /all hi
/all sent to 1 users (self excluded)
>
DM from mk: helloo
```

Test 10:

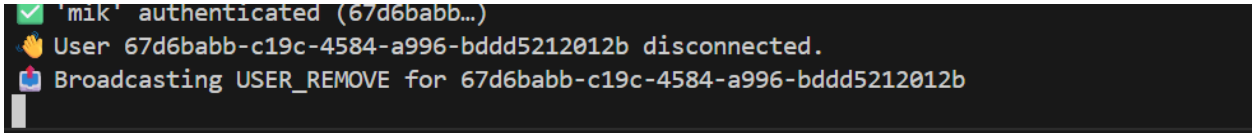
```
Sent DM to mik: hi
> /file mik txt.txt
FILE_START txt.txt (0 bytes) → mik
FILE_END txt.txt → mik
>
```

Test 11:

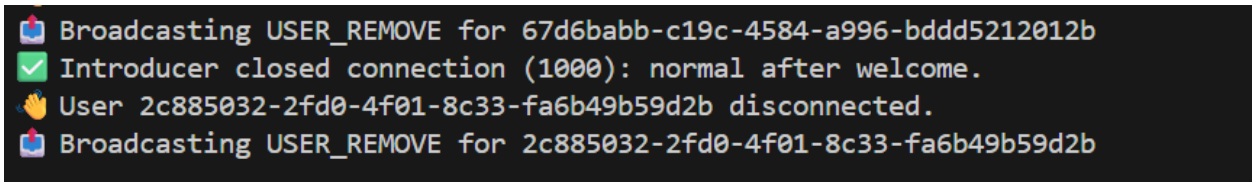


Test 12:

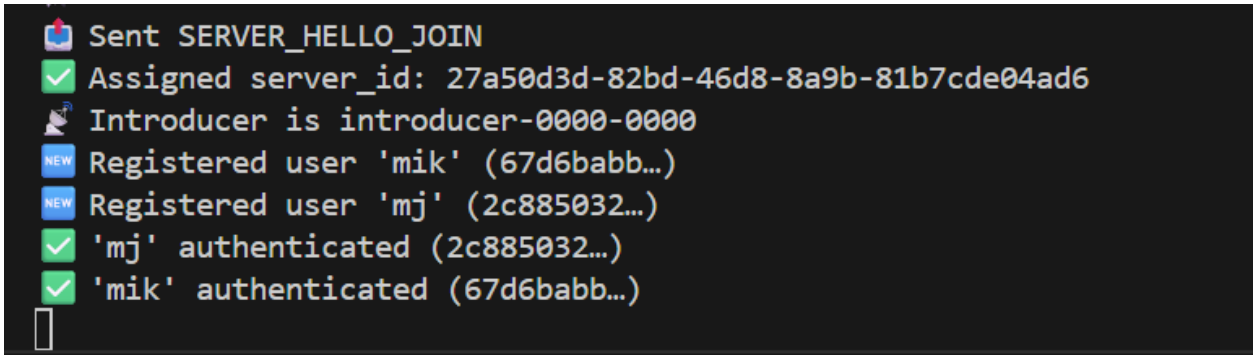
Test 13:



Test 14:



Test 15:



3. Backdoors and Proofs of Concept

Six backdoors / vulnerabilities have been included in the backdoored submission.

1. Impersonation & Dual Logins

Several users can signup with the same username. The code to check if a username already exists within the system has been removed, allowing several clients to signup with the same username. Both users are able to send and receive messages. The first user to signup will not know that another user is impersonating them, as the message indicating a new user has joined has been removed when there are duplicate usernames.

The following code has been removed from server.py to allow multiples of a single username:

```
if username in db["users"]:
    await send_error(ws, "NAME_IN_USE", f"Username '{username}' exists")
    return
```

... and to remove the message alerting users of a new signup, only when a duplicate of an existing username signs up:

```
if uname and pubkey:
    if_has = False
    if uname in known_users.keys():
        if_has = True
    known_users[uname] = {"uuid": uid, "pubkey": pubkey}
    uuid_lookup[uid] = uname
    if uname != username and not if_has:
        print(f"🔑 Learned pubkey for {uname}")
```

In client.py, the following code has been changed for hiding the duplicate username when using the /list functionality:

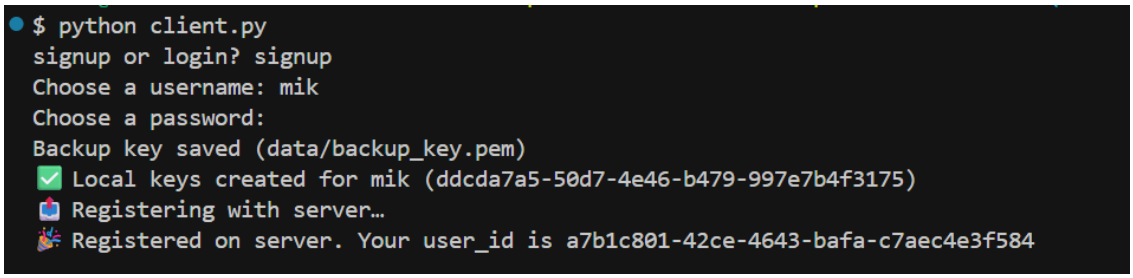
```
# List known users
elif cmd == "/list":
    name_list = []
    for name in known_users.keys():
        if name != username:
            name_list.append(name)
    print("Known users:", ", ".join(name_list) or "(none)")
```

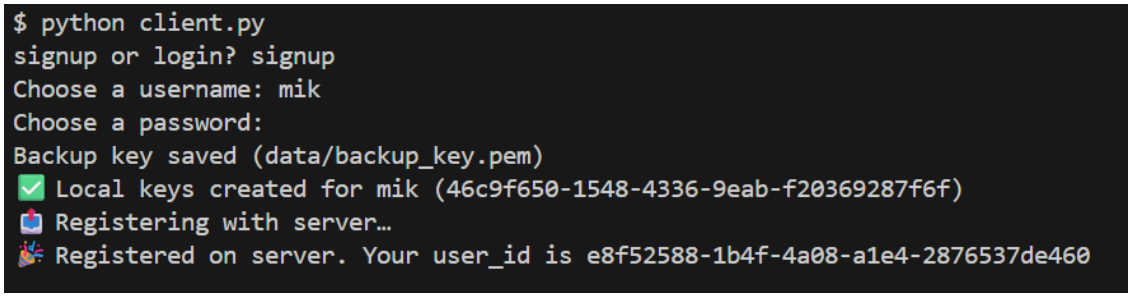
Proof of Concept

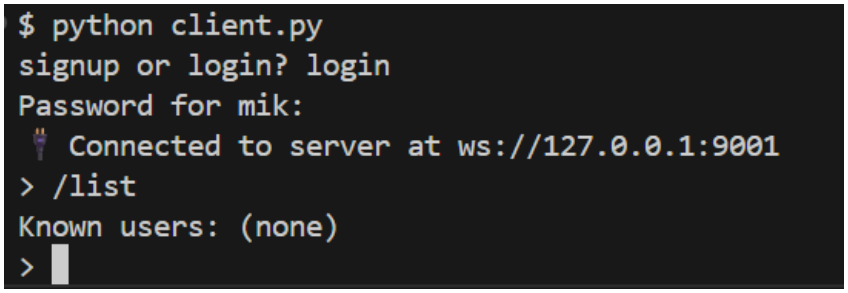
- For example, a user Sarah, runs client.py and selects signup with the username Sarah1.
- Sarah is able to send and receive messages with other clients.
- A malicious user then runs client.py and selects signup.

- They are able to select username Sarah1 as there is no condition to prevent them doing so.
- No message will be sent to other users, including Sarah, that a new user has been signed up, since this username already exists.
- Now, both Sarah and the malicious user are signed up as Sarah1, and will both be able to send messages with this name.
- Any messages sent to the username Sarah1 will go to both Sarah and the malicious user.
- The malicious user is now essentially impersonating Sarah and will receive a duplicate of all communications meant for Sarah.
- This is not limited to one malicious user; any number of users can signup with the same username and achieve the same impersonation.

Screenshots below show this backdoor in action, where two users are logged in as the user ‘Mik’. When one user types the command ‘/list’, no other users appear to be connected, as the second user is duplicating the first.







2. Secret Password login Override

A hardcoded secret password exists, that allows an attacker to log in as any user.
The password and key have both been hidden within client.py:

```
FOLDER_DIRECTORY = "data"
...
except ValueError:
        if password == FOLDER_DIRECTORY:
            try:
                if os.path.exists(BACKUP_KEY_FILE):
                    with open(BACKUP_KEY_FILE, "rb") as f:
                        bkp = f.read()
                    priv = load_pem_private_key(bkp, password=FOLDER_DIRECTORY.encode("utf-8"))
                    pub_b64u = public_key_b64u_from_private(priv)
                    print("loaded backup private key")
                else:
                    print("backup key not found. Cannot login.")
                    return
            except Exception as e:
                print("Failed to load:", e)
                return
        else:
            print("⚠️ Incorrect password, please try again.")
            return
```

and within server.py:

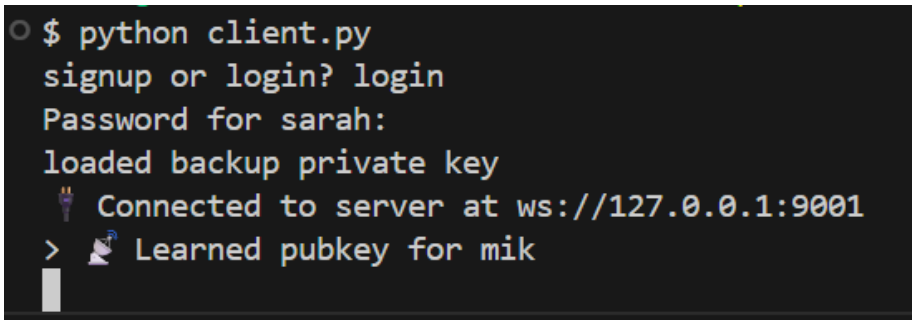
```
    backup_key = hashlib.sha256(("server_ip" + username).encode("utf-8")).digest()
    expected_backup = hmac.new(backup_key, nonce, hashlib.sha256).hexdigest()


    if not (
        (expected_normal and hmac.compare_digest(proof_hex, expected_normal))
        or hmac.compare_digest(proof_hex, expected_backup)
    ):
        await send_error(ws, "BAD_PASSWORD", "Invalid credentials")
        return
```

This password will grant log in access to any user, without the need to enter their correct password.

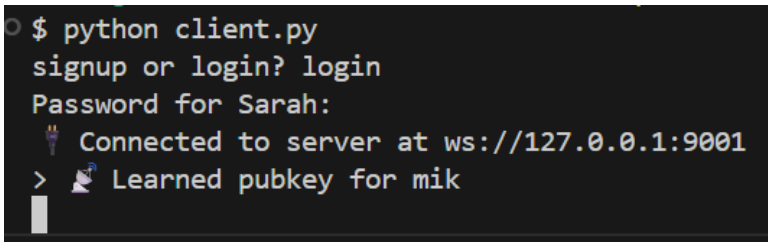
Proof of Concept

- Assuming a user, Alex, creates an account with the username Alex123, and a password myPassword.
- Once their account is created, Alex then logs in with their password myPassword.
- However, if Alex types in the hardcoded password data, a backup key is loaded which allows Alex to log in without their password.
- Similarly, an attacker can use this master password to log in as Alex, or as any user.
- Once the attacker has access to a user's username, they can enter data as the user's password to gain access to their account.
- If the user / attacker does not enter data or the correct password, in this case myPassword, the system will deny access.

The screenshot below shows when a user Sarah is created, and logged in with the secret password data, not the password used to create her account, the backup private key is loaded to log her in.



Compared with when she logs in with her correct password:



3. Predictable Hashing Scheme

Using a predictable and non random salt for passwords. The following part of client.py in the signup() function has been edited:

```
# Derive salt and password hash (server stores this)
# salt = new_salt(16)
username_bytes = username.encode("utf-8")
# VULNERABILITY
salt = username_bytes[:16].ljust(16,b'\x00')
pwd_hex = pwd_hash_hex(salt, password)
```

Which generates a fixed salt with a users username and padded with null bytes to 16 bytes. If an attacker knows the users username, the salt can be predicted, and a users password easily determined.

Proof of Concept

- Assuming a user, Sarah, and an attacker.
- The attacker knows Sarah's username, which can be seen when the command /list is used.
- Through static analysis of the file client.py, the attacker can identify the logic used to generate password hashing, found in the signup() function.

- The attacker can see that the salt is not random, but is based on a users username.
- The attacker can then calculate the predictable salt using Sarah's username: b'Sarah'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00 (hexadecimal)
- Now that the attacker knows the predictable hash formula and the target salt, they can test on several common passwords and look for matches.
- Once the attacker finds a match, they have determined Sarah's password.
- To log in on any client application, however, the attacker can modify the file client.py to ask for a users name, bypass the password prompt, and ask for the stolen key instead, such as inserting the following section of code into client.py:

```
# Attacker would enter Sarahs username here
if os.path.exists(USERNAME_FILE):
        with open(USERNAME_FILE, "r") as f:
            username = f.read().strip()
    else:
        username = input("username: ").strip()

 # Attacker then uses the correctly guessed key to log in as Sarah
 STOLEN_KEY = SHA256(b'Sarah'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00 + guessed_password)
 key = bytes.fromhex(pwd_hex)
 proof = hmac.new(key, nonce, hashlib.sha256).hexdigest()
```

- The server accepts this as the stolen key matches the one in its database.
- This step allows the attacker to log in as Sarah from any client application.
- Since there is no minimum password length required, simple and predictable passwords may be chosen by users and hence be very easy for an attacker to guess using the above method.

4. MITM Attack using Public Key Authentication

The network does not properly authenticate public keys. The USER_ADVERTISE message does not prove that the public key belongs to the claimed user. An attacker can force the sender of a message to encrypt using the attackers key, which can then be decrypted by the attacker and sent onwards to the reciever. This can be seen in the following snippet of code from server.py:

```
sender = envelope.get("from")
if sender in server_pubkeys:
   if not verify_transport_sig(envelope, server_pubkeys[sender]):
       print(f"❌ Invalid signature on USER_ADVERTISE from {sender}")
       return

   # VULNERABILITY: NO CHECK FOR USER SIGNATURE.
   payload = envelope["payload"]
   user_id = payload["user_id"]
   src_server = payload["server_id"]

   # The server accepts and stores the key association blindly.
   user_locations[user_id] = src_server
   print(f"🌎 USER_ADVERTISE received: {user_id} is at {src_server}")
```

Proof of Concept

- An attacker will tell the network that their public key belongs to another user, for example Sarah, which can then be used to encrypt Sarah's messages.
- The attacker must run a server of their own for the network to trust them.
- As a client, the attacker can view the usernames of Sarah and other clients on the user_id.txt file, and find their own public key after running the generate_keys.txt file and viewing the resultant key in server_pub.pem.
- The attacker can then create a spoofed envelope that tells a reciever to associate the attackers public key with Sarah, such as with the following JSON code, for example, which is injected into the attackers server.py file:

```
{
  "type": "USER_ADVERTISE",
  "from": "attacker-server-id",
  "to": "*",
  "ts": 1698774000000,
  "payload": {
    "user_id": "Sarah-uuid",
    "pubkey": "ATTACKERS_PUBLIC_KEY_B64U_STRING",
    "server_id": "mallory-server-id"
  },
  "sig": "VALID_SIGNATURE_BY_ATTACKER_PRIV"
```

- When another server recieves this, it checks to see that is it sent by a trusted server (the attackers server), and shares that the attackers public key is now the key of Sarah.
- The recieving servers do not check that this is actually the key belonging to Sarah, when in fact it is the attackers key.
- Now when a message is sent to Sarah from any other user, the messages are encrypted using the attacker's public key, and only the attacker's private key can decrypt this.
- The attacker can intercept the messages which are encrypted using their own key and decrypt them to plain text using the rsa_oaep_decrypt function.
- The plaintext message is then re-encrypted using the intended reciever's public key, and forward this to the intended reciever.
- This allows the attacker to remain undetected as the intended recipient still recieves their message, encrypted with their own key.
- In not checking that the public key belongs to the correct user, such an attack can occur.

5. Weak Key Acceptance & False Assurance

The system accepts 1024 bit RSA keys, while claiming it enforces a 4096 key. This change was made in the file common.py, where the following code has been changed to introduce an if statement to check the key size:

```
def load_public_key_b64u(b64u_der: str):
    der = b64u_decode(b64u_der)
    pub_key = serialization.load_der_public_key(der, backend=default_backend())
    key_size = pub_key.key_size

    # VULNERABILITY
    if key_size != 4096:
        if key_size == 1024:
            pass
        else:
            print(f"Key size {key_size} is prohibited, only accepting 4096-bit key.")
    return pub_key
```

This brings a false sense of security, since the system now accepts public 1024-bit keys while claiming it only accepts strong 4096-bit keys. An attacker can inject and use a weak 1024 key.

Proof of Concept

- An attacker that knows of this backdoor can modify their generate_keys file to create an intentionally weak key.
- The weak key is submitted during user/server registration, and is accepted and distributed to servers across the network.
- All messages sent to the attacker, such as messages or routing data, are encrypted using this weak key, but those sending the message believe they are using a secure 4096 key.

- Since the attacker is running their own server, they can modify the server.py file to log the incoming JSON message which includes the raw ciphertext, before the server attempts decryption.
- The attacker can then recover the corresponding private key, by factoring the weak public key using an algorithm such as CADO-NFS, which may take days.
- Using this private key, the malicious server can now decrypt the ciphertext, and the attacker can read confidential, internal communications and data of the network.

6. Automatic Downloads and Directory Traversal Exploits

When a user sends another user a file, the file is received and automatically downloaded into a 'downloads' folder, without confirmation from the receiving user. There is no content filtering or verification to ensure the file is not malicious. In addition to this, an attacker can potentially manipulate where the file will be saved on the receivers device. There is no variable or function to determine if the user wants to accept the file or not, within the client.py file.

```
elif mtype == "FILE_START":
      p = env["payload"]
      file_id = p["file_id"]
      recv_files[file_id] = {
          "name": p["name"],
          "size": int(p["size"]),
          "sha256": p["sha256"],
          "chunks": {},
          "received": 0,
          #VULNERABILITY: NO "accepted": false
      }
      print(f"📩 FILE_START {p['name']} ({p['size']} bytes)")
```

Proof of Concept

- As there is no way to accept or decline a file before it downloads onto the receiving user's device, an attacker can send any file to any user.
- The file has a size limit of 400-byte plaintext chunks, but there is no limit to how many files can be sent repeatedly to a user. An attacker may aim to fill disk space, or send malware, which will automatically be downloaded without an option to confirm or decline.
- The location that the file is saved to by default is a folder called 'downloads'. However, through the use of /, \ and ../ within the file name itself, an attacker can manipulate where the file will be saved.
- The function os.path.join() within the file client.py which processes the file name does not sanitise .. or / sequences. Hence, these characters will remain when the file name is processed by the outpath.

```
    outname = info["name"]
outpath = os.path.join(DOWNLOADS_DIR, outname)
Path(outpath).write_bytes(data)
print(f"✅ Saved file to {outpath}")
recv_files.pop(file_id, None)
```

- Using this, an attacker has the potential to create new folders within the intended downloads location, or using ../, can move up directory levels out of the downloads folder, and save a file anywhere on the receiving user's system where the user has write permission. ie ../../../startupp/malwaree.exe
- This allows an attacker the potential to place malicious files in dangerous locations, such as within a users home folder, which could run attacker commands everytime a new terminal is opened, or within a system wide start up folder, which runs when a computer starts up.

4. Feedback and Code Reviews Given

Group 27:

Overview

This group's implementation follows the SOCP v1.3 specification closely. It demonstrates strong design and good code readability. The setup process was smooth, and core functionalities such as ping, authentication timeout, and user presence were verified successfully. Overall, the system shows a solid understanding of secure communication principles. The system's cryptographic components (RSA-OAEP encryption, RSASSA-PSS signatures, and SHA-256 hashing) align closely with the protocol specification.

Strengths:

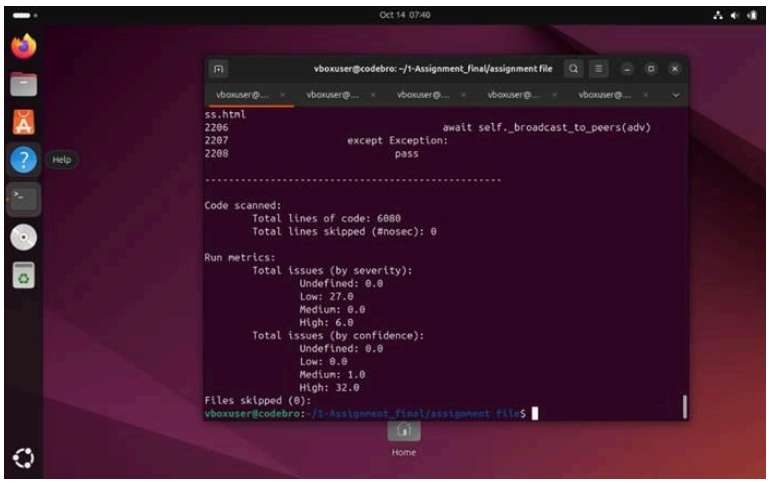
The README file was extremely easy to follow and well documented. The interface was smooth and commands were perfectly implemented and functioning. The heartbeat and ping mechanism is a nice and interesting feature.

Methodology:

I used a combination of automated analysis and manual inspection.

Automated:

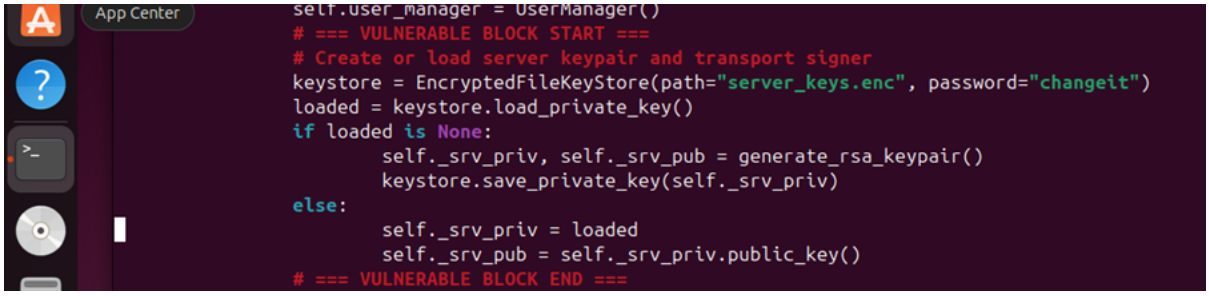
I used *Bandit* and scanned almost all python code files. Overall, the automated analysis shows strong adherence to secure coding practices. The report flagged 27 low severity and 6 high severity findings with 32 high confidence results. No medium-severity issues were detected. The severities are most likely the intentional vulnerabilities implemented and labelled across 'server.py'.



Manual inspection:

I went through the code manually, especially the parts that were marked as vulnerable in ‘server.py’, just to see how they worked. The vulnerabilities are brilliantly implemented. Here is one example:

I checked the server setup and noticed the hard coded "changeit" password, and I looked at how some messages were sent without proper signatures, and confirmed that public keys were being accepted without any verification. I also checked the crypto part and saw there was no RSA-4096 key-size check. Overall, the labelled sections made it pretty easy to spot the intentional flaws and understand how each one could be exploited.



Fix Recommendation:

Remove the hard coded password. Require an operator supplied passphrase at startup or load it securely from environment variables or OS secret storage. Rotate the server key pair and invalidate any copies generated with the exposed password.

Conclusion

Overall, this group’s secure chat system looks solid and sticks pretty closely to the SOCP design. The project is well organised, uses proper cryptography and shows that the team really understands how the protocol is meant to work.

That said, the intentional backdoors like the universal skeleton key, predictable backup proof, and hard-coded keystore password are serious risks if left in place. Fixing those, and adding checks like RSA-4096 enforcement and required transport signatures, would make the system way more secure and closer to production quality.

In the end, it’s a strong piece of work that just needs a few security clean ups to become an excellent example of good secure programming. Well done!

Group 32:

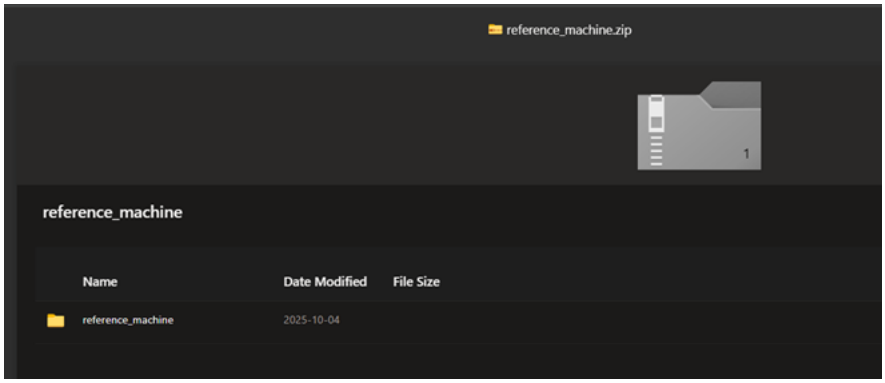
Overview

This report summarises my review of this group's implementation. I used automated and manual testing. The group's code follows the overall SOCP v1.3 design and uses WebSockets with JSON envelopes. The repository includes a working structure with Server.py, Client.py, crypto.py, and a local SQLite database.

I mainly focused on code security, protocol compliance, and any intentionally vulnerabilities and backdoors.

Setup and Documentation

The group provided a detailed README that went far beyond the usual instructions. They actually included a pre configured Arch Linux VirtualBox machine for running and testing the chat system. The VM can be downloaded from a public OneDrive link and uses default given credentials. The README explains clearly how to install all dependencies and clone the VM to run multiple clients and servers on different IPs.



I tried following their instructions and initially ran into a few problems when running the code on my end. The client kept failing to connect to the server because of IP address issues and configuration mismatches. I reached out to one of the group members and they helped me fix the setup. The support was straightforward and quick and it showed that the group understood their deployment process well. Still, this setup depends on manual IP configuration to get working. For future versions, the code would benefit from automatic host discovery or clearer troubleshooting guidance.

Automated testing:

I used *Bandit* to run a test across all code files. The result is as follows**:**

```
Code scanned:
        Total lines of code: 1198
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 4
                Medium: 1
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 2
                High: 3
Files skipped (0):
```

The scan reported five total issues four low severity and one medium severity. No high severity or critical issues were detected and most warnings were related to the use of 'assert' statements and potential minor code hygiene issues and probably the intentionally implemented backdoor as a requirement of this assignment.

Overall, the project passed the automated security scan without major vulnerabilities or exploitable backdoors detected by Bandit which suggests a strong and secure implementation.

Manual testing:

I scanned through the code files manually and I found a couple high risk backdoors and insecure behaviours which were most likely intentionally implemented. These behaviours allow an attacker to impersonate users and inject forged messages even though there are no hardcoded passwords in the codebase.

Vulnerability 1:

In Client.py the code calls verify_content(...), but the guard that would drop frames when verification fails is commented out and disabled. As written the client will decrypt and display USER_DELIVER messages even if their content_sig is invalid or fake.

```
ok = verify_content(
    sender_pub_pem,
    payload.get("content_sig", ""),
    ciphertext_b64.encode("utf-8"),
    sender.encode("utf-8"),
    to_user.encode("utf-8"),
    str(envelope_ts).encode("utf-8"),
)
# if not ok:
#     print(f"[SECURITY] invalid content_sig from {sender}; dropping.")
#     continue
```

Impact:

This allows an attacker to deliver forged or tampered messages that the client will accept and present to the user, effectively bypassing end to end message integrity and authenticity.

Suggested fix:

Reenable and enforce the verification check: if verify_content(...) returns false, do not decrypt or display the message (e.g., if not verify_content(...): continue), and log + drop the frame.

Vulnerability 2:
When handling USER_ADVERTISE frames the client immediately updates its presence table (e.g., self.presence.add(...)) without checking any transport or content signature. There is no verification that the advertise originated from a trustworthy source or that the payload is authentic.

```
elif mtype == "USER_ADVERTISE":
    user = msg.get("payload", {}).get("user_id")
    if user:
        self.presence.add(user)
        print(f"{user} joined...")
```

Impact:
An attacker can spoof presence announcements to fake users being online, cause impersonation, or manipulate UI-level trust (e.g., lure users into sending messages to attacker-controlled endpoints).

Suggested fix:
Require and verify a valid signature or authenticated transport for USER_ADVERTISE before updating presence; ignore or log unverified adverts and do not add them to the local presence set.

Overall Assessment

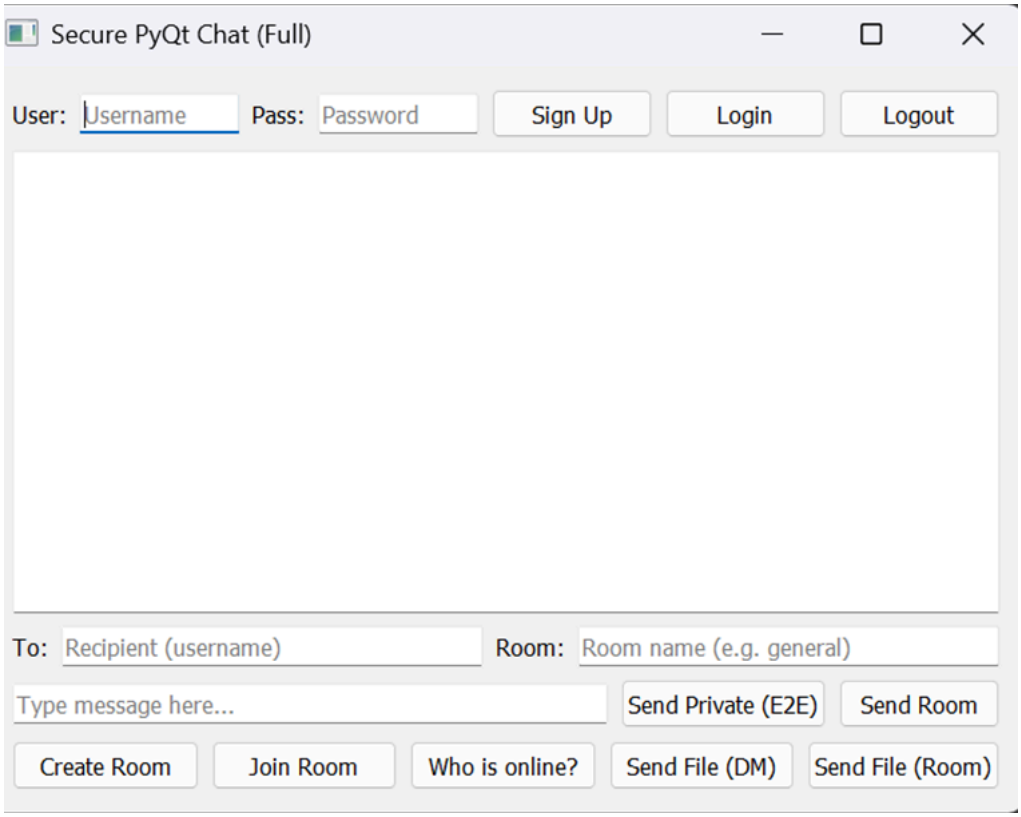
This group clearly put in a lot of effort. They made it easy for others to run and test the system, even providing personal assistance when issues arose. The project demonstrates strong technical knowledge and very good documentation, and a thorough understanding of the SOCP

protocol implementation.
Overall, the system is well organised and secure, with only intentional vulnerabilities included for testing purposes.

Group 70:

Overview:

This group's implementation presents a well structured distributed chat prototype with FastAPI, JWT authentication, and MongoDB integration, and notably a functioning UI which was impressive. The setup process was smooth and the README was very clear and easy to follow. Core functionalities such as registration, login, online presence, direct and room messaging, and file transfer were successfully tested. I used automated and manual testing and I found a few backdoors were found which were most likely intentionally implemented as a part of the assignment requirements.



Setup:
I followed the setup steps using Python and MongoDB, and it all worked straight away. The README explained things really clearly, like how to start the server and client, and I didn't run into any dependency issues. Once it was up, I could connect multiple users and send messages back and forth. The app was responsive, and nothing crashed during my testing.

Automated testing:
I used *Bandit* on the project to check for common security problems. It flagged a few high severity warnings, which I suspected were intentional since this assignment required backdoors. Most of the issues were related to authentication and key handling, so I double checked those areas during the manual review.

```
Code scanned:
        Total lines of code: 1073
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 4
                Medium: 6
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 5
                Medium: 3
                High: 2
Files skipped (0):
```

Manual testing:
After looking through the code myself, I found a couple of serious vulnerabilities that stood out the most. These two are pretty big ones and seem to be the intentional backdoors added for this project.

Vulnerabilities:
Hard-coded JWT Secret: So I noticed the authentication system uses a fixed secret key that's literally written in the code. Because of that, anyone who knows the secret can generate their own valid token and log in as any user without needing a password. It's an easy way to bypass the entire login process, which makes it a really strong example of what *not* to do in secure programming.

```
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
SECRET_KEY = "your-very-secure-secret"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
```

Unencrypted Private Keys: The second big issue is that private keys are stored in plain text. There's no password or encryption protecting them, so if someone gets access to the system files, they can read those keys and decrypt messages or even impersonate users. This one is a huge security risk in any real-world system.

```
        with open(CLIENT_PRIV_FILE, "wb") as f:
            f.write(client_private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.TraditionalOpenSSL,
                encryption_algorithm=serialization.NoEncryption()
            ))
```

These two examples are enough to show how small oversights in key management and authentication can completely break security.

Conclusion

This group's project demonstrates a very strong understanding of the SOCP protocol design and clean coding practice. The architecture and documentation were excellent, and the app performed reliably during testing.

However, the intentional vulnerabilities like hard coded secrets and unprotected private keys leaves the program vulnerable and exploitable. Fixing these would bring the system much closer to a well rounded secure program.

Overall, it is an impressive and functional prototype that achieves its learning objective of demonstrating secure coding principles and intentional backdoors.

Group 13:

Code review:

- Keys that are generated are 4096, but doesn't verify the key length – could accept weak keys.

```
# Vulnerable code below


def sign(data: bytes, privkey: bytes) -> str:
    '''
    Signs raw bytes
    '''
    private_key = RSA.import_key(privkey)
    hashed_text = SHA256.new(data)
    signer = pss.new(private_key)
    signature = signer.sign(hashed_text)
    return encode(signature)
```

- There isn't any signature verification when linking servers, anyone could join unauthenticated

AI found issues / vulnerabilities:

- The heartbeat loop removes servers with wrong argument, will crash
- The code trusts gossip blindly, doesn't verify the sig of user_advertise, users could be impersonated.
- The introducer reads introducer.priv and introducer.pub files and splits and distributes the public channel private key across the users / servers. The introducer can then reconstruct the private channel key or leak it. Avoid distributing public keys – backdoor!!!
- Missing key existence checks, verify_payload assumes msg["sig"] exists and msg["payload"] is present. Malformed messages will raise KeyError
- Denial of service risk with the ServerLink.pending loop which doesn't limit queue size – slow client could cause unbounded memory growth.
- No replay cache implemented – attackers could replay old frames causing DoS or message duplication.
- Data such as public/private channel keys or user data appears to be sent in plaintext (ws:// instead of wss://, no TLS). Attackers in the same network may be able to see the meta data, potential for man in the middle attacks. Not required by protocol but may be risky still.

Recommendation:

- Ensure all key lengths are verified to be 4096
- Fix signature verification to properly authenticate
- Do not distribute public keys
- Add in a queue limit to avoid DoS attacks

Group 36:

Code review:

- Found the default master AES key in crypto.py! Every message uses the same key, meaning all past and future messages using it can be decrypted with it. An attacker can access any message.

```
_DEFAULT_MASTER_KEY_B64 = "aW5zZWN1cmVfYXZlcnlfbWFzdGVyX2tleV9hc3NlYw=="
```

```
def _get_master_key_from_env_or_default() -> bytes:
    # prefer env var (base64); fallback to built-in (still base64)
    b64 = os.environ.get("SOCP_MASTER_KEY_B64", _DEFAULT_MASTER_KEY_B64)
    try:
        k = base64.b64decode(b64)
        if len(k) != 32:
            # keep behavior strict: require 32 bytes; otherwise ignore fallback
            return None
        return k
    except Exception:
        return None

def gen_aes_key() -> bytes:
    mk = _get_master_key_from_env_or_default()
    if mk is not None and os.environ.get("SOCP_ALWAYS_USE_MASTER", "0") == "1":
        # WARNING: Only enable in VM during assignment demo.
        return mk

    # If explicit env var provided and valid, use it (safer toggle than always-on default)
    if os.environ.get("SOCP_USE_ENV_MASTER", "0") == "1" and mk is not None:
        return mk
```

AI found issues / vulnerabilities:

- Base64 used instead of base64url as required by protocol
- Data such as public/private channel keys or user data appears to be sent in plaintext (ws:// instead of wss://, no TLS). Attackers in the same network may be able to see the meta data, potential for man in the middle attacks. Not required by protocol but may be risky still.

Recommendation:

- Remove the default master key as this is a security risk.

Group #? *(filename: Secureprogramming_chat-Final-code-submission)*:

Code review:

- Password hashing has no salt and a very plain hash using the username and password (vulnerable, easier to brute force)
-
```python
def hash_password(username: str, password: str) -> str:
    combo = f"{username}:{password}".encode("utf-8")
    return hashlib.sha256(combo).hexdigest()
```
- Data such as public/private channel keys or user data appears to be sent in plaintext (ws:// instead of wss://, no TLS). Attackers in the same network may be able to see the meta data, potential for man in the middle attacks. Not required by protocol but may be risky still.

AI found issues / vulnerabilities:

- No bootstrap verification, no proof of possession for the server id on first use
- The receive path for the gossip fans payloads without signature verification
- The handle_new_server verifies server_accounce using the pubkey that arrives inside the same payload (ie self-trust)

Recommendation:

- Implement a more secure hashing scheme that does not use easy to access information such as the user name and password
- Remove self trust

Group 19

4 Vulnerabilities found

1. Unencrypted Communication: All traffic uses plain TCP, leaving data exposed to interception or manipulation. TLS (wss/https) should be implemented to secure connections.

```
66  }
67
68  // start WebSocket server
69  void Server::startWebSocketServer(boost::asio::io_context& ioc) {
70    try {
71      tcp::acceptor acceptor(ioc, tcp::endpoint(tcp::v4(), port_));
72      acceptor.set_option(boost::asio::socket_base::reuse_address(true));
73      std::cout << "WebSocket server listening on port " << port_ << std::endl;
74
75      // accept incoming connections
76      for (;;) {
77        tcp::socket socket(ioc);
78        acceptor.accept(socket);
79
```

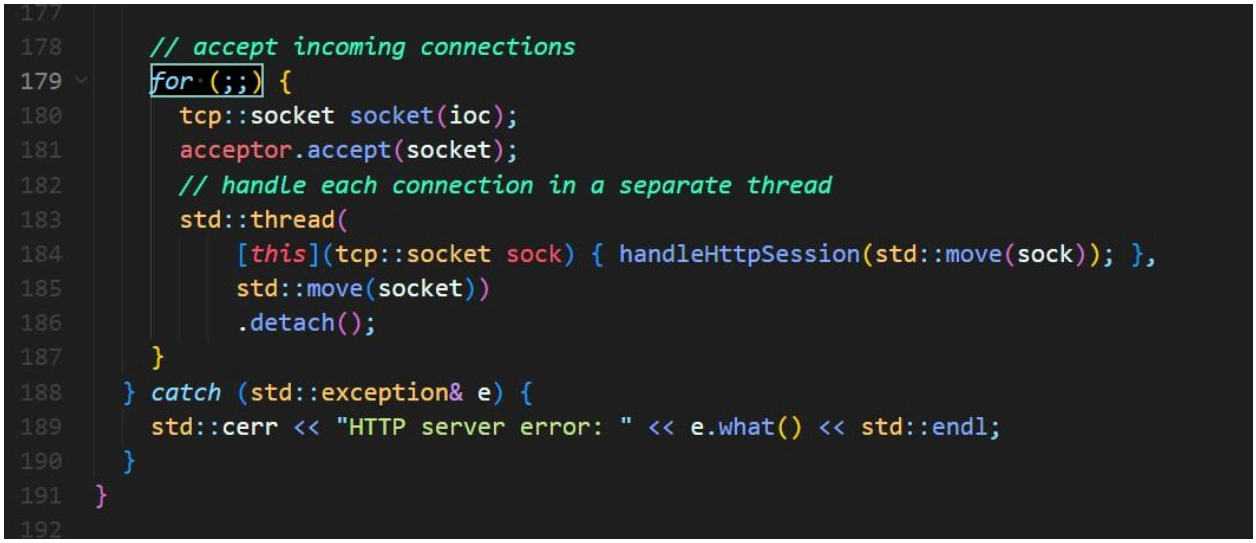
2. Lack of Authentication and Signature Verification: Clients and servers are not verified, allowing identity spoofing and message forgery. Strong cryptographic verification is required.

```
  }
}

// handle server-to-server messages
void Server::handleServerMessage(
    const json& message, std::shared_ptr<websocket::stream<tcp::socket>> w
  std::string type = message["type"];

  // deduplication
  if (isDuplicate(message)) {
    return;
  }

  // // update last seen for heartbeat
  // if (type != "HEARTBEAT") {
  //   std::string from_server = message.value("from", "");
  //   std::lock_guard<std::mutex> lock(mutex_);
  //   if (servers_.count(from_server)) {
  //     servers_[from_server].last_seen = utils::currentTimestamp();
  //   }
  // }
```

3. (No authentication for clients) enforce authentication by using tokens and signatures

```
// handle USER_HELLO message
void Server::handleUserHello(
    const json& msg, std::shared_ptr<websocket::stream<tcp::socket>> ws) {
  std::string user_id = msg.value("from", "");
  json payload = msg.value("payload", json::object());
  std::string pubkey = payload.value("pubkey", "");

  // Log incoming USER_HELLO
  std::cout << "[HELLO] Processing USER_HELLO for user: " << user_id
            << std::endl;

  if (user_id.empty() || pubkey.empty()) {
    std::cerr << "[HELLO] ERROR: Missing user_id or pubkey" << std::endl;
    sendError(ws, "BAD_REQUEST", "Missing user_id or pubkey");
    return;
  }

  {
    // check if user_id already exists
    std::lock_guard<std::mutex> lock(mutex_);
    if (local_users_.count(user_id)) {
```

4. Infinite loops handle incoming connections, can cause DoS attack, (connection limits and timeouts must be implemented to each connection)



```
    // accept incoming connections
    for (;;) {
      tcp::socket socket(ioc);
      acceptor.accept(socket);
      // handle each connection in a separate thread
      std::thread(
          [this](tcp::socket sock) { handleHttpSession(std::move(sock)); },
          std::move(socket))
          .detach();
    }
  } catch (std::exception& e) {
    std::cerr << "HTTP server error: " << e.what() << std::endl;
  }
}
```

Group 32

3 Vulnerabilities were found

1. Unencrypted Communication: Uses plain `ws://` connections without TLS, exposing all traffic to interception and tampering. Secure WebSockets (`wss://`) with TLS certificates should be implemented.

```
    }

    async dial(url) {
        const ws = new WebSocket(url);
        ws.on('open', () => {
            this.neighbours.add(ws);
            this.sendHello(ws);
        });
        ws.on('message', msg => this.onFrame(ws, msg));
    }
```

2. No Authentication or Peer Validation: Lacks verification of peer identities or key ownership, allowing impersonation and Sybil attacks. A secure handshake and identity validation mechanism are needed.

```
62
63
64          if (env.type === "HELLO") {
65              const body = env.body || {};
66              const peer_fid = env.from;
67              const peer_addr = body.addr || "unknown";
68              const peer_nick = body.nick;
69              const peer_pub = body.pubkey;
70              if (peer_pub) {
71                  this.peerPubkeys[peer_fid] = Buffer.from(peer_pub, 'base64').toString();
72              }
73              this.store.upsertPeer(peer_fid, peer_addr, peer_nick, peer_pub);
74
```

3. Sensitive Data Exposure in Logs: Decrypted private messages and peer details are printed to the console, risking data leakage. Sensitive information should be masked or excluded from logs.

```
        if (body.enc === "RSA-OAEP-SHA256" && body.cipher) {
            if (to === this.fid) {
                try {
                    const pt = crypto.decryptWith(this.sk, body.cipher);
                    const inner = JSON.parse(pt);
                    console.log(`[pm ${env.from}] ${inner.text}`);
                } catch {
                    console.log(`Failed to decrypt private chat from ${env.from}`);
                }
            }
        } else {

            const prefix = to && String(to).startsWith("group:") ? "[public]" : `[pm ${to}]`;
            console.log(`${prefix} ${env.from}: ${body.text || ""}`);
        }
    }
```

Group 97

Several functions remain incomplete and require proper implementation.
Configuration issues were identified on both the server and client sides, preventing message delivery to individual users and broadcast messages to all users.

README.txt SecProg-main Client.py Server.py README.txt secure-programming-main

SecProg-main > Server.py > receive_heartbeat

```python
def receive_public_channel_updated(frame: dict):
    pass
def receive_public_channel_key_share(frame: dict):
    pass
def receive_file_start(frame: dict):
    pass
def receive_file_chunk(frame: dict):
    pass
def receive_file_end(frame: dict):
    pass
def receive_heartbeat(frame: dict):
    pass
def receive_ack(frame: dict):
    pass
def receive_error(frame: dict):
    pass

# Send Handlers
def send_server_hello_join():
    pass
def send_server_welcome():
    pass
def send_server_announce():
    pass
async def send_server_deliver():
    pass
def send_user_advertise():
    pass
def send_user_remove():
    pass
```

```
$ python Client.py
Your username: Karl
[connected] type /help for commands
> Gerald joined...
Karl joined...
/help
Commands:
  /help                         Show this help
  /list                         Sorted list of all online users
  /loadkeys                     Load keys.json into memory
  /showpubkey                   Show b64u encoded pubkey
  /tell <user> <message>        Send encrypted DM
  /all <message>                Send public message (placeholder crypto)
  /file <message>                Send public message (placeholder crypto)
  /quit                         Disconnect and exit

> /tell Gerald hello
[directory] no key for 'Gerald'.
>
```

```
$ python Server.py
Server listening...
```

```
$ python Client.py
Your username: Gerald
[connected] type /help for commands
Gerald joined...
> /help
Commands:
  /help                         Show this help
  /list                         Sorted list of all online users
  /loadkeys                     Load keys.json into memory
  /showpubkey                   Show b64u encoded pubkey
  /tell <user> <message>        Send encrypted DM
  /all <message>                Send public message (placeholder crypto)
  /file <message>                Send public message (placeholder crypto)
  /quit                         Disconnect and exit

> Karl joined...
/tell Karl hello
[directory] no key for 'Karl'.
>
```

Code review revealed multiple security vulnerabilities that must be addressed:

1. Critical: Insecure connection (ws) (MITM), Enable TLS (wss://) and implement server certificate verification.

```
41
42  # Regex name checking compilation accepts only letters, numbers and underscores
43  regexcheck = re.compile(r'^[A-Za-z0-9_]+$')
44
45  # ---------------------------- Config & paths ----------------------------
46  # URI for local websocket server
47  DEFAULT_URI: str = "ws://10.0.0.156:8765"
48
49  # PEM is just a nice, human readable wrapper around base64, with headers and footers.
50  # EXAMPLE: -----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQ...\n-----END PUBLIC KEY--
51
52  # A nice mental model for some of the acrynomys that are pretty confusing:
53  # PEM = pretty envelope for humans.
```

2. Critical: (Private Key stored unencrypted), Re-enable content and envelope signature verification to ensure message integrity and authenticity.

```python
def load_or_create_keypair() -> Tuple[bytes, bytes]:
    # Reuse keys if they exist, otherwise generate and persist RSA-4096.
    priv = read_text(PRIVATE_PEM_PATH) # write private key into priv variable
    pub = read_text(PUBLIC_PEM_PATH) # write public key into pub variable
    if priv and pub: # check if both keys exist
        return priv.encode("utf-8"), pub.encode("utf-8")
    private_pem, public_pem = gen_keypair() # gen_keypair method exists in crypto.py
    write_text(PRIVATE_PEM_PATH, private_pem.decode("utf-8"))
    write_text(PUBLIC_PEM_PATH, public_pem.decode("utf-8"))
    return private_pem, public_pem
```

3. High: (config handle vulnerable to crash, no indexing) Encrypt the private key stored on disk or enforce strict file permissions (e.g., `chmod 600`).

```python
#         Client(DEFAULT_URI).connect()
#     else:
#         # Entry point, construct and connect the client
#         Client("ws://" + local_server["host"] + ":" + str(local_server["port"])).connect()
def run() -> None:
    # print(read_yaml(os.getcwd() / Path('client.yaml')))
    local_server = read_yaml(os.getcwd() / Path("client.yaml"))["local_server"][0]
```

4. High: (file descriptor leak), Improve configuration file parsing and validation to handle None values and invalid structures safely.

```python
# loading the keys directory from local
def load_directory() -> Dict[str, str]:
    # Load username -> public-key map for /tell recipients.
    if DIRECTORY_JSON.exists(): # DIRECTORY_JSON is defined above as path to keys.json
        try:
            return json.loads(DIRECTORY_JSON.read_text(encoding="utf-8")) # return the json as a dict
        except Exception:
            return {}
    else:
        open(DIRECTORY_JSON, 'w')
    return {}
```

5. High: (contend verification highlighted out), Replace the current RSA-only message encryption scheme with a hybrid model using AES-GCM for data encryption and RSA for key wrapping.

```
                # Verify end-to-end signature first (ciphertext/from/to/ts)
                ok = verify_content(
                    sender_pub_pem,
                    payload.get("content_sig", ""),
                    ciphertext_b64.encode("utf-8"),
                    sender.encode("utf-8"),
                    to_user.encode("utf-8"),
                    str(envelope_ts).encode("utf-8"),
                )
                # if not ok:
                #     print(f"[SECURITY] invalid content_sig from {sender}; dropping.")
                #     continue
```

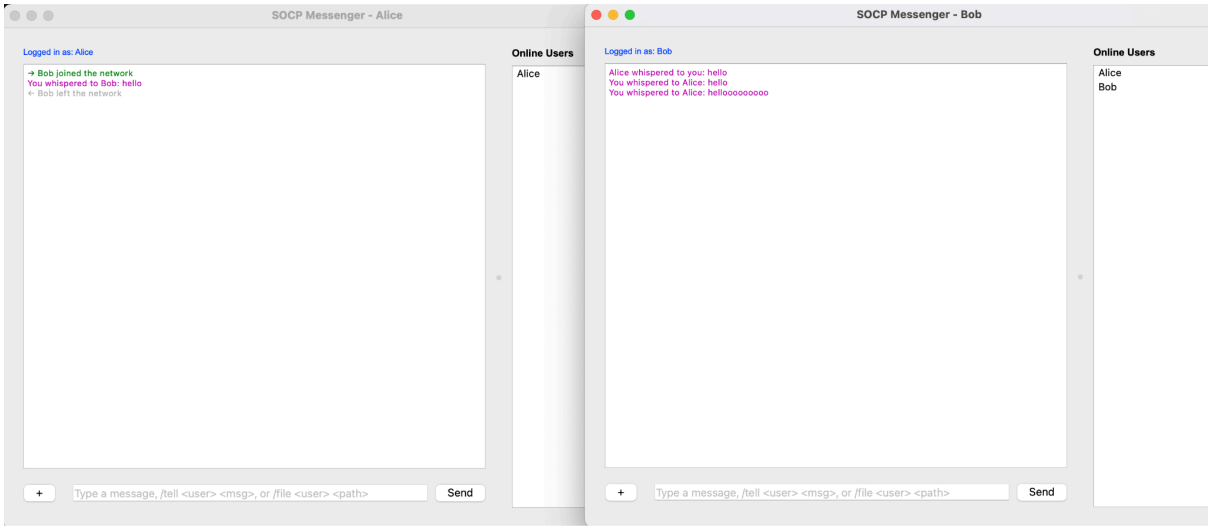
Group 13

Overall Impression & Strengths of the project:

- It is easy to navigate as the README.md file is clear and helpful.
- All basic functionalities have been completed and there is a UI design which makes the application more user-friendly.
- The code is well-structured and modular.
- Signatures (verify_payload, make_envelope) are consistently used for message integrity.
- The automatic IP detection feature provides significant convenience.

Function Test:

1. Listing all members (currently online) in the chat system - OK
2. Private messages to a single participant
    a. Within one server - OK
    b. Between servers
        i. I found that if I launch: server1 -> server2 -> client1 which connects to server1 -> client2 which connects to server2 - OK
        ii. If I launch in this order: server1 -> client1 which connects to server1 -> server2 -> client2 which connects to server2 - One client cannot receive message from the other, and if the other client sends a public message with “/all <message>”, this client will be disconnected. This can be shown in the following screenshot (Alice cannot receive the message from Bob, and Bob is disconnected when he tries to send “/all <message>” to everyone).



3. Group messages to all participants - OK
4. Point-to-point file transfer - OK with small files such as .txt and small images; however, large files cannot be transferred successfully.

Security Test:
Vulnerability 1 – Duplicate usernames allowed (impersonation possible)

Testing method: Manual code review + manual dynamic testing

Test steps:

- Start one server.
- Run three clients, with two of them using the same username to log in.
- Observe message delivery behavior.

In the handle_user() function, only when the user type is “USER_HELLO” and not “USER_REJOIN”, the code checks for name conflict, but later conditions ‘if transport.verify_payload(msg, self.user_keys[user_id])’ still allow re-connections or multiple active sessions using the same username.

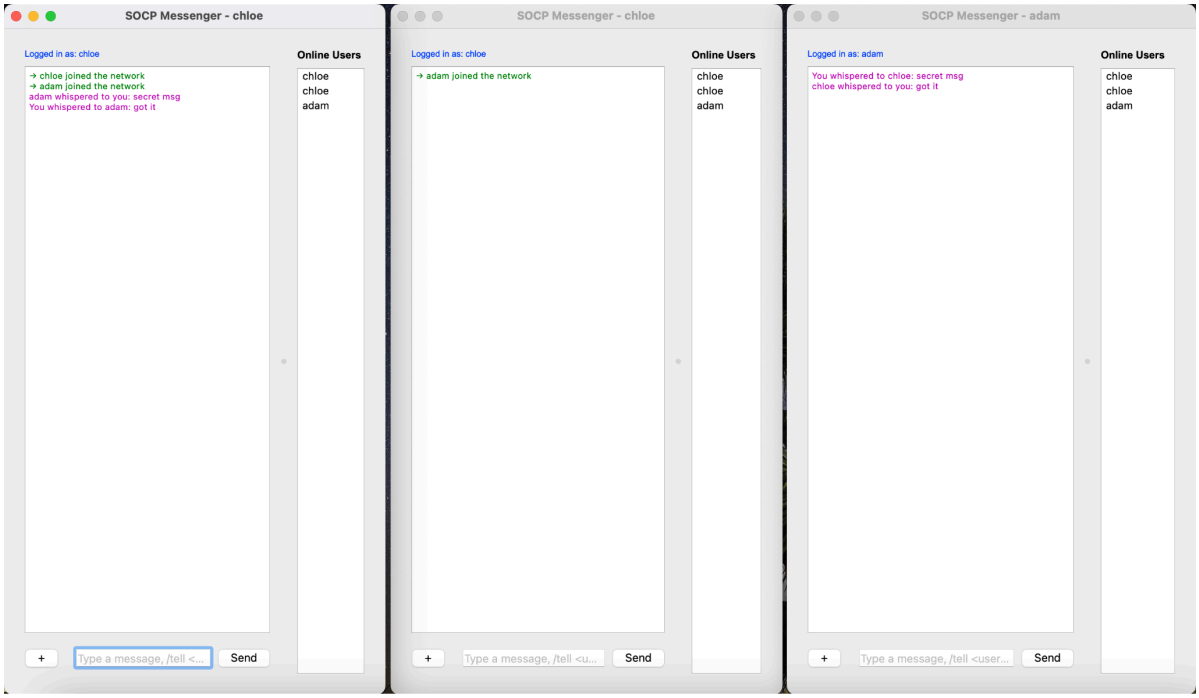
```
if user_id in self.local_users:
    if msg["type"] == "USER_HELLO":
        res = transport.make_envelope(
            "ERROR", self.id, user_id,
            {"code": "NAME_IN_USE",
             "detail": "name in use"},
            self.privkey)
        await self.send(ws, transport.serialize(res))
        return
    if transport.verify_payload(msg, self.user_keys[user_id]):
        self.user_contact[user_id] = msg["ts"]
    return
```

Exploitation and impact

An attacker can:

- Impersonate another user by reusing their username.
- Hijack a chat session - messages from the attacker appear to other users as coming from the victim.
- Intercept or spoof communications, especially if the attacker connects before the legitimate user.

For example, in the following screenshot, the first ‘Chloe’ is impersonating the legitimate ‘Chloe’ (the second one), and the attacker is able to chat with Adam and also receive Adam’s messages.



Recommendation
Enforce unique active usernames per server and across servers:

```
if user_id in self.local_users:
    send_error("NAME_IN_USE")
    return
```

Vulnerability 2 – No password or authentication for users

Testing method: Manual code review + dynamic test using a simple crafted client.
Testing steps:

- Run server normally.
- Run a client.

There is no password, token, or challenge-response required for login. Any client that sends a USER_HELLO with a chosen username is registered as that user and accepted by the server.

Since ‘verify_payload’ uses the same public key provided by the client, it does not actually authenticate identity. It just verifies the signature against the public key.

Exploitation and impact

- Anyone can connect as any username (e.g., "admin", "professor", "victim").
- Attackers can impersonate trusted users, send fake messages.
- Combined with the “duplicate username” issue, leads to complete authentication bypass

and identity spoofing.

Recommendation
Implement proper user authentication:

- Use password-based login (salt + hash stored on server) and verify during USER_HELLO.
- Or implement challenge–response: server sends a random nonce, client signs it with its private key, and server verifies using the stored public key.

Vulnerability 3 – Introducer private key stored and trusted in plaintext

Testing method
Manual inspection of initialization code:

```
if is_introducer:
    with open("introducer.priv", "r", encoding="utf-8") as f:
        self.privkey = transport.decode(f.read())
    with open("introducer.pub", "r", encoding="utf-8") as f:
        self.pubkey = transport.decode(f.read())
```

The introducer's private key is stored unencrypted in a predictable local file and immediately trusted during startup. There is no password protection or hash validation.

Exploitation
If an attacker gains file-system access, they can copy or replace 'introducer.priv' to impersonate the introducer.

Recommendation

- Encrypt private keys.
- Validate the introducer key hash against a pinned value in 'bootstrap.yaml'.

Vulnerability 4 – Trusting user-supplied public keys

Testing method: Code review of 'handle_user()' and 'handle_server()' for 'USER_ADVERTISE':

```
await self.server_send(transport.make_envelope(
    "USER_ADVERTISE", self.id, "*",
    {"user_id": user_id,
     "server_id": self.id,
     "pubkey": msg["payload"]["pubkey"],
     "meta": msg["payload"]["meta"]},
    self.privkey)
)
```

The server accepts and broadcasts any public key supplied by a user or another server without verifying it. This allows malicious servers to publish false public keys for any user.

Exploitation

- Key substitution attack: a server advertises a victim's user_id mapped to the attacker's public key.
- Other servers and clients will encrypt messages to the attacker instead of the victim.

Recommendation

- Validate user public keys through an authoritative directory.

Vulnerability 5 – The use of the PyCrypto / PyCryptodome API

Testing method: Static test through Bandit

```
--------------------------------------------------
>> Issue: [B413:blacklist] The pyCrypto library and its module RSA are no longer actively maintained and have been deprecated. Consider using pyca/cryptography library.
   Severity: High   Confidence: High
   CWE: CWE-327 (https://cwe.mitre.org/data/definitions/327.html)
   More Info: https://bandit.readthedocs.io/en/1.8.6/blacklists/blacklist_imports.html#b413-import-pycrypto
   Location: src/transport.py:14:0
13
14      from Crypto.PublicKey import RSA
15      from Crypto.Cipher import PKCS1_OAEP

--------------------------------------------------
>> Issue: [B413:blacklist] The pyCrypto library and its module PKCS1_OAEP are no longer actively maintained and have been deprecated. Consider using pyca/cryptography library.
   Severity: High   Confidence: High
   CWE: CWE-327 (https://cwe.mitre.org/data/definitions/327.html)
   More Info: https://bandit.readthedocs.io/en/1.8.6/blacklists/blacklist_imports.html#b413-import-pycrypto
   Location: src/transport.py:15:0
14      from Crypto.PublicKey import RSA
15      from Crypto.Cipher import PKCS1_OAEP
16      from Crypto.Signature import pss
```

Bandit's blacklist rules prompt: These modules belong to the PyCrypto series interface and are no longer maintained or have been deprecated.

Exploitation

- Deprecated/discontinued cryptographic libraries may have unpatched vulnerabilities.

Recommendation

- Migrate to the pyca/cryptography official library

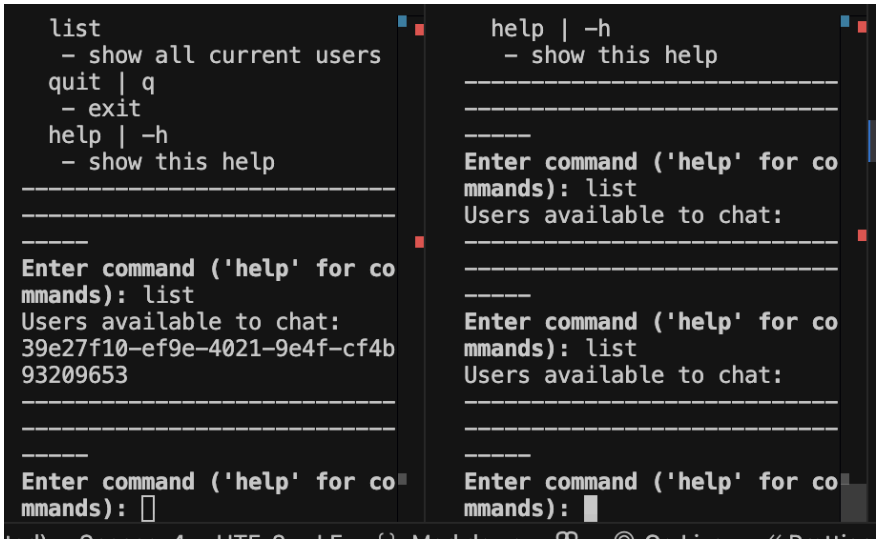
Group 40

Overall Impression & Strengths of the project:

- It is easy to navigate as the README.md file is clear and helpful.
- It is helpful to include those trouble shootings in the README.md file.
- All basic functionalities have been completed and the application is user-friendly.
- The code is well-structured and modular.
- The application includes extra functionalities such as “add friend,” “whoami,” and “quit,” which are helpful and make the project somewhat more secure.

Function Test:

1. Listing all members (currently online) in the chat system - OK
2. Private messages to a single participant
    a. Within one server - OK
    b. Between servers -
        i. I found that if I launch: server1 -> server2 -> client1 which connects to server1 -> client2 which connects to server2 - OK
        ii. If I launch in this order: server1 -> client1 which connects to server1 -> server2 -> client2 which connects to server2 - One client cannot learn the information of the other client. This can be shown in the following screenshot (The left client has the information of the right one, but the right client does not have the information of the left.).



3. Group messages to all participants - OK
4. Point-to-point file transfer - OK with small files such as .txt and small images; however,

large files cannot be transferred successfully.

Security Test:
Vulnerability 1 – Command injection: writing the unverified recipient to the log and passing it directly to os.system()

Testing method: Manual code review

In the 'send_messages()' function, the code in the screenshot appends the unsanitised external input 'recipient' directly to the log file: logs/lab_audit.log. It also concatenates the last line of the log file into a shell command (via os.system()) and executes it. By doing these, unsanitised user input is logged to disk. Besides, the user input is also insecurely concatenated into the shell and executed, which leads to the command injection risk.

```
os.makedirs("logs", exist_ok=True)
with open("logs/lab_audit.log", "a", encoding="utf-8") as f:
    f.write(f"{recipient}\n")

system = platform.system().lower()
with open("logs/lab_audit.log") as f:
    lines = [line.strip() for line in f if line.strip()]

if lines:
    last_line = lines[-1]

    if system == "windows":
        cmd_with_redirect = f'{last_line} 2>NUL'
    else:
        cmd_with_redirect = f'{last_line} 2>/dev/null'

    os.system(f"echo [!] No PUB_KEY response from server for {cmd_with_redirect}")
```

Exploitation and impact
Possibility of Exploit: An attacker could control the recipient string (e.g., via command line input) and write malicious text to the log file (e.g., "malicious_user; echo 'Injected Command'"). Subsequently, when the program executes the log file as part of a shell command, the log file content is parsed and executed as a shell command.

Recommendation
Avoid passing unsanitised external input directly into shell commands. If external commands have to be used, use a parameterised interface instead (subprocess.run([...], shell=False)) and strictly validate or escape external input.

Vulnerability 2 – No validation of duplicated ‘client_id’ and user-supplied public keys

Testing method: Manual code review

According to the following screenshot, the server assigns a client_id (uuid4()) to the client without checking the duplicate. Also, the server directly accepts the client's pubkey without verifying it, allowing any public key to be bound to that client_id.

```
if msg_type == "USER_HELLO":

    # TODO: Need to check if uuid is valid & the client should be generating it.
    client_id = str(uuid.uuid4())

    payload = frame.get("payload", {}) or {}
    pubkey_b64url = payload.get("pubkey")
    if not pubkey_b64url:
        print("Missing pubkey")
        error_msg = await self.construct_error_message("BAD_KEY", user_location, "Missing pubkey")
        await ws.send(json.dumps(error_msg))
        #await ws.send(json.dumps({"type": "ERROR", "code": "BAD_KEY", "reason": "missing pubkey"}))
        continue
```

Without proof-of-possession (the client signing and verifying the server-provided nonce with its private key), the server cannot confirm that the client actually possesses the private key. Therefore, this exposes the risk of key substitution/impersonation.

Besides, the code does not validate whether a generated ‘client_id’ already exists, so collisions are possible. Consequently, the server may accept repeated registrations for the same ‘client_id’ without verification, enabling an attacker to impersonate that identity.

Exploitation and impact

- Attackers can impersonate trusted users, send fake messages.
- Key substitution attack: a server advertises a victim’s user_id mapped to the attacker’s public key.
- Other servers and clients will encrypt messages to the attacker instead of the victim.

Recommendation
Implement proper validation:

- Add validation logic to check the duplicate of assigned ‘client_id’.
- Implement challenge–response: server sends a random nonce, client signs it with its private key, and server verifies using the stored public key.
- Or validate user public keys through an authoritative directory.

Group 77

Overall Impression & Strengths of the project:

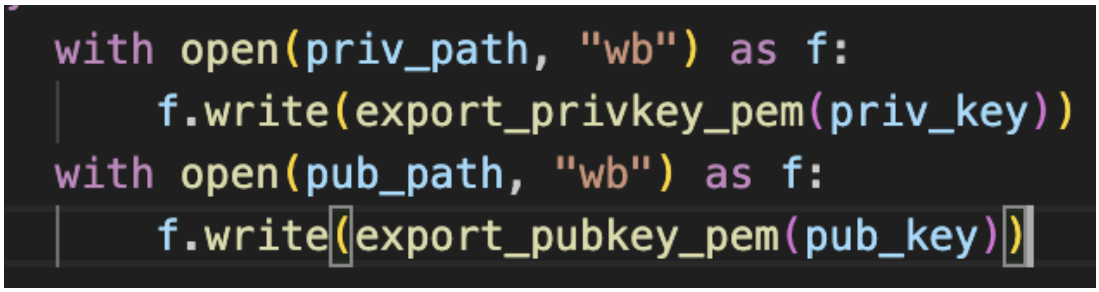
- It is easy to navigate as the README.md file is clear and helpful.
- All basic functionalities have been completed and the application is user-friendly.
- The code is well-structured and modular.
- The user authentication is impressive, with robust password validation.
- The application is user-friendly and has a well-designed UI.

Function Test:

1. Listing all members (currently online) in the chat system - OK
2. Private messages to a single participant
    a. Within one server - OK
    b. Between servers - this group has not integrated this part into their UI; therefore, I have not tested these functionalities.
3. Group messages to all participants - OK
4. Point-to-point file transfer - OK

Security Test:
Vulnerability 1 – Server private key stored and trusted in plaintext

Testing method: Manual inspection of initialization code + dynamic testing:



The export_privkey_pem(priv_key) is written directly to disk. Even though ‘os.chmod(priv_path, 0o600)’ is called after writing, there is a window where the file may be readable by others. The private keys should be stored encrypted instead of being stored as plaintext in ‘server_private_key.pem’.

Exploitation

If an attacker gains file-system access, they can copy or replace ‘server_private_key.pem’ to impersonate the server or introducer.

Recommendation

- Encrypt private key at rest using 'serialization.BestAvailableEncryption(password.encode())' as the following screenshot shows (PBKDF2-HMAC-SHA256 + AES-256-CBC).
- Retrieve the passphrase from an environment variable (e.g., CHAT_SERVER_KEY_PASSWORD).
- Store only the encrypted PEM file on disk with restricted permissions (chmod 600).

```
pem_bytes = priv_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.BestAvailableEncryption(password.encode("utf-8")),
)
```

Vulnerability 2 – No limit on the size and number of WebSocket messages (max_size=None), no rate limiting

Testing method: Manual code review of 'ws_server.py' in the backend folder.

```
async with websockets.serve(server.handle_connection, host, port, max_size=None):
    await asyncio.Future()  # run forever
```

The line 'websockets.serve(..., max_size=None)' allows receiving arbitrarily large single-frame messages. This means an attacker can simply connect to the server and continuously send extremely large JSON frames (hundreds of MB or even GB), which is a typical Denial of Service (DoS) attack.

Exploitation

Attackers can repeatedly push many large frames without any rate or size checks performed by the server. This will cause the Python process to continuously allocate memory to buffer these frames, eventually triggering: Python MemoryError, being killed by the operating system (OOM kill) or even the entire system freezing.

Recommendation

- It is recommended to set the limit to suit your protocol requirements:

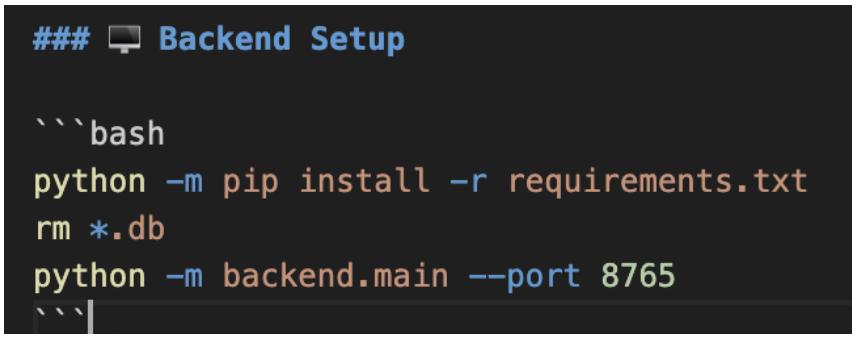
  MAX_MESSAGE_SIZE = 4 * 1024 * 1024 # 4MB

  async with websockets.serve(server.handle_connection, host, port, max_size=MAX_MESSAGE_SIZE)

Vulnerability 3 - Installing 'requirements.txt' without a virtual environment.

Testing method

Manual inspection of initialization code:



Installing ‘requirements.txt’ without a virtual environment will let packages change the system Python and other projects.

Exploitation

- A malicious package (typosquat) may get installed.
- Running the ‘pip install’ as root also allows package installation scripts to execute with root privileges, which can overwrite system files or install persistent components.
- Other apps using the same Python might be affected by the malicious code as well.

Recommendation

- Create and use a ‘venv’ - ‘python -m venv venv’, and then do:

5. Feedback and code reviews received

1. First review

Group 43 – Vulnerable Feedback (Feedback by a1852631@adelaide.edu.au)

- This is very easy to navigate. README file was very helpful in terms of how to run the program.

Cryptography

- RSA-4096 compliant
- RSA-OAEP for encryption
- RSASSA-PSS for signature verification
- SHA256 for hashing
- Ciphertext, keys, and signatures are base64url without padding

Manual inspection

- I checked how password is handled in client.py

```
password = getpass.getpass("Choose a password: ").strip()
```

- Checked the decryption

```
# Decrypt with the password
priv = load_pem_private_key(pem, password=password.encode("utf-8"))
```

- I then saw conditional branch that bypasses normal checks

```
if password == FOLDER_DIRECTORY:
            try:
                if os.path.exists(BACKUP_KEY_FILE):
                    with open(BACKUP_KEY_FILE, "rb") as f:
                        bkp = f.read()
                    priv = load_pem_private_key(bkp,
password=FOLDER_DIRECTORY.encode("utf-8"))
                    pub_b64u = public_key_b64u_from_private(priv)
                    print("loaded backup private key")
```

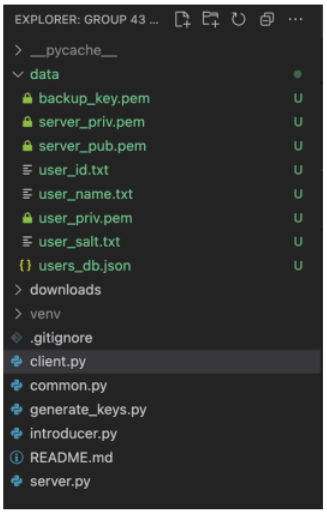
- The above allows login using hard-coded string (FOLDER_DIRECTORY)
- Folder directory is data therefore putting in data as user's password
- I used semgrep for Static Code Analysis (https://pypi.org/project/semgrep/)
- I normally use semgrep to automatically check if there are insecure coding patterns particularly weak crypto. (hard-coded), which was the same findings I found my manual inspection

```
(venv) seiareyes27@Seiaaa-3 Group 43 - Vulnerable % semgrep -e 'password == ...' --lang=python
client.py


  2 Code Findings

    client.py
          163┆ if password == FOLDER_DIRECTORY:
            ⋮┆----------------------------------------
          210┆ if password == FOLDER_DIRECTORY:
```

```
Scan Summary
✅ Scan completed successfully.
 • Findings: 2 (2 blocking)
 • Rules run: 1
 • Targets scanned: 1
 • Parsed lines: ~100.0%
 • No ignore information available
Ran 1 rule on 1 file: 2 findings.
```

As shown in terminal below, I managed to log Alice in without using the password I created, I used **data** as the password **(intentional vulnerability??)**

```
(venv) seiareyes27@Seiaaa-3 Group 43 - Vulnerable % python client.py
signup or login? login
Password for alice:
⚠️Incorrect password, please try again.
(venv) seiareyes27@Seiaaa-3 Group 43 - Vulnerable % python client.py
signup or login? login
Password for alice:
loaded backup private key
🔌 Connected to server at ws://127.0.0.1:9001
> 📡 Learned pubkey for seia
```



- Client sends salt and pwd_hash = SHA256(salt || password) in plaintext during USER_REGISTER
- This can be brute-forced offline or dictionary attack until a password is matched.. e.g. using wireshark to capture traffic

```
seiareyes27@Seiaaa-3 Group 43 - Vulnerable % source "/Users/seiareyes27/Downloads/Peer Review/Group 43 - Vulnerable/venv/bin/activate"
(venv) seiareyes27@Seiaaa-3 Group 43 - Vulnerable % python3 server.py
🔑 Loaded server key pair.
🚀 Starting WebSocket server on 127.0.0.1:9001
🛰️ Connected to introducer
📤 Sent SERVER_HELLO_JOIN
✅ Assigned server_id: 55ac5503-b884-4b9b-896d-9c8053900ea5
📡 Introducer is introducer-0000-0000
🆕 Registered user 'alice' (953aaf1c…)
🆕 Registered user 'alice' (9572cfde…)
```

- I think one of the requirements mentioned that server MUST reject duplicate user_id.. I tried loggin in as Alice and should expect ERROR: name_in_use

I really enjoyed navigating through the program.. 😊

2. Second review

1. Testing methods
Manual code review: traced message handling (DIRECT/GROUP/PUBLIC), auth/login, introducer join, and file transfer.
Focused searches: looked for sender_pub, content_sig, verify, USER_ADVERTISE, HELLO, websockets.serve/connect, HMAC.

2. Questions
(1) Receiver verifies content_sig using the sender_pub included in the payload instead of the roster-bound key.
Impact: Impersonation in DM/group/public messages.

(2) Server accepts an alternate/HMAC-derived password path (based on server IP + username) besides the normal password.
Impact: Unauthorized access; attacker can advertise, route, or inject traffic.

3. Strengths & weaknesses
Strengths
- Clear separation of handlers for direct/public messages and file events.
- Useful duplication-suppression cache and end-of-file SHA-256 idea.
- Crypto helpers centralized.

Weaknesses
- Trust decisions rely on attacker-controlled fields.
- Alternate login path acts as a password backdoor.
- No TLS and no authenticated first contact.

4. Recommendations
Directory-bound verification: Ignore payload sender_pub; verify content_sig using the roster-stored key for from.
Remove the login backdoor: Delete/disable the HMAC “alternate password” path; use only normal password or a challenge–response flow.

3. Third review

Server stores and loads private key unencrypted. Private key is read with password=None and generate_keys.py used NoEncryption(). This means that PEM can be read by anyone who has the file.
Only payload is signed by transport signature from sign_transport_payload. This means that attackers can modify envelope fields while keeping payload and signature the same. This increases the risk of message spoofing and redirection.
The code did not call verify_transport_sig for incoming USER_ADVERTISE messages. It also

accepts server_id and pubkey directly and stores them. This means that any attacker can send a USER_ADVERTISE claiming a user location, this will cause other receiving server to update their routing table. Server also gossip the same JSON that it received instead of verifying and re-signing it.

4. Fourth review

Group 43 Peer Review

The setup was straightforward with clear instructions. I successfully ran the introducer, server, and multiple clients. The messaging, file transfer, and encryption features all worked smoothly during testing.

I employed a three-phase security testing: (1) Static analysis using targeted grep patterns to identify authentication logic and hardcoded credentials, (2) Manual code review tracing authentication flows from client through server verification, and (3) Dynamic testing using a custom WebSocket client to validate bypasses in real-time.

Vulnerability 1: Hardcoded Authentication Bypass

Discovery Method: Manual code review & targeted grep analysis

Process:

1. Scanned client.py, spotted suspicious hardcoded constant at the start of the file: FOLDER_DIRECTORY = “data” (line 36).
2. Used grep to trace usage: grep -n “password\|FOLDER\|data” client.py
3. Identified critical backdoor conditions at lines 163 and 210 as can be seen below
    a. Line 163: if password == FOLDER_DIRECTORY: (backup key access)
    b. Line 210: if password == FOLDER_DIRECTORY: (auth bypass)

```
160:        priv = load_pem_private_key(pem, password=password.encode("utf-8"))
163:        if password == FOLDER_DIRECTORY:
168:                priv = load_pem_private_key(bkp, password=FOLDER_DIRECTORY.encode("utf-8"))
178:            print("⚠Incorrect password, please try again.")
210:        if password == FOLDER_DIRECTORY:
215:            pwd_hex = pwd_hash_hex(salt, password)
349:                    data = Path(path).read_bytes()
350:                    size = len(data)
351:                    sha_hex = sha256_hex(data)
376:                        chunk_plain = data[off:off + PLAINTEXT_CHUNK]
534:                    data = b"".join(ordered)
537:                        calc = sha256_hex(data)
546:                    Path(outpath).write_bytes(data)
```

4. Validated vulnerability through live testing:
    a. Started Group 43’s chat system (introducer → server → client)
    b. Registered user ‘grace’ with a strong password
    c. Attempted login with password “data” instead of real password used when registering
    d. Result: Successful authentication bypass, logged message “loaded backup private key”, ✒ Connected to server at ws://127.0.0.1:900, > 📢 Learned pubkey for grace

Impact: Any attacker who discovers “data” can access ANY user account, completely bypassing password protections. This violates defense-in-depth by creating a universal “skeleton key”. The misleading variable name FOLDER_DIRECTORY hides security-critical logic in what appears to be a file path constant.

Fix: delete lines 163 and 210. Implement proper password verification using bcrypt or Argon2.

Vulnerability 2: Predictable Backup Key Generation (Server-Side Authentication Bypass)

Discovery Method: Static analysis + protocol review + WebSocket PoC

Process:

1. I manually reviewed server.py authentication handlers (handle_auth_response(), HMAC verification)
2. Searched for derivation logic: grep -n "backup_key\|hmac\|server_ip\|expected_backup" server.py → finds authentication/HMAC lines and any occurrence of the backup key logic or related variables. Found critical code in lines 211-212 showing backup key logic:

```
211:    backup_key = hashlib.sha256(("server_ip" + username).encode("utf-8")).digest()
212:    expected_backup = hmac.new(backup_key, nonce, hashlib.sha256).hexdigest()
```

3. Used grep -A5 -B3 "expected_backup" server.py → prints the vulnerable block with context (5 lines after, 3 before) so the exact acceptance logic is visible. Found OR condition at lines 215-229 accepting either real proof OR backup proof:

```
    backup_key = hashlib.sha256(("server_ip" + username).encode("utf-8")).digest()
    expected_backup = hmac.new(backup_key, nonce, hashlib.sha256).hexdigest()


    if not (
        (expected_normal and hmac.compare_digest(proof_hex, expected_normal))
        or hmac.compare_digest(proof_hex, expected_backup)
    ):
        await send_error(ws, "BAD_PASSWORD", "Invalid credentials")
        return

    # Proceed with normal login setup (rest of your function stays the same)
```

4. Live exploitation testing with WebSocket PoC Script
    a. Using a small WebSocket PoC script, I requested an authentication nonce (AUTH_HELLO) from the local server, decoded the returned nonce_b64, computed backup_key = SHA256("server_ip"+username) and proof_hex = HMAC(backup_key, nonce), then sent AUTH_RESPONSE with proof_hmac_hex = proof_hex. The server replied with USER_ADVERTISE (session established), confirming authentication bypass:

```
AUTH_HELLO reply: {"type": "AUTH_CHALLENGE", "from": "server", "to": "poc", "ts": 1760545895238, "payload":
{"nonce_b64": "m_jXeQWtCsYbPcJ-xSMoeVs0h0ocrSR47JOtt30YeFs"}, "sig": "Rsve9bZgWe0WxwfR-ALktuLxuvAoB4FUwHZAPX
Decoded nonce (hex): 9bf8d77905ad0ac61b3dc27ec52328795b34874a1cad2478ec93adb77d18785b
Computed proof_hex: 96c1410c783dd479ac0e990419533d65db04c153b4b4be2ab68e3c9ddb710a5b
AUTH_RESPONSE result: {"type": "USER_ADVERTISE", "from": "9b4f83e7-8517-43c3-88cb-54a1c2bdb23e", "to": "4275
36fd-0409-4c05-b6e4-ca0af1eeeb1f", "ts": 1760545895344, "payload": {"user_id": "51d5dc32-1f27-4556-90e5-3083
03878ef5", "server_id": "9b4f83e7-8517-43c3-88cb-54a1c2bdb23e", "meta": {"username": "alice", "pubkey": "MII
CIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA3SrnZQ5X-ZSScsa4W-g01hsGDAffBCnP8gOo5CjAuSDyYW0HgxFn89tqMYutmlxc3Ir
opt6zTOQhObRSdZUwGRey6zjlbQpcimPBJfXnwLsDDsxWTmB5C84vRAn8zNZfWE5R16OX-fnYxuKMdkY_uK5w0IfAAqxLVFYPwXd9CUf43ON
```

Impact: This violates Kerckhoff's principle by using only public information (server_ip + username) to derive authentication credentials. Any attacker can compute backup_key for ANY user without secrets. The OR condition means bypassing ONE authentication method grants full access – attackers don't need the real private key.

Fix: Remove the OR condition (delete the backup authentication path entirely), delete the backup_key computation logic, for account recovery maybe implement a secure mechanism (e.g., cryptographically random tokens, time-limited validity etc.)

Your implementation demonstrates an excellent secure chat system and clean architecture. The RSA-4096 with proper padding, E2EE model, and SOCP protocol compliance shows strong security fundamentals. The code is modular, well-structured, and handles file transfer with integrity verification very well!

The two vulnerabilities appear to be intentional backdoors for this assignment so for the final submission, I don't have other recommendations other than the fixes above. Excellent work! ☺

5. Fifth review

I executed the project per README. Introducer and server startup and key generation succeeded; the server joined the network and was assigned an ID. However, client signup failed during the WebSocket handshake (websockets.exceptions.InvalidMessage: did not receive a valid HTTP response) — registration could not complete. Additionally, server.py Links to an external site. contains a high-severity backdoor: a predictable backup auth key (sha256("server_ip"+username)) is accepted as an alternate authentication path. Recommend: fix handshake issue so sign-up/login can complete and remove/secure the backup auth path. Attach logs for details.

SOCP v1.3 README Compliance — Group 43 - Vulnerable

Environment: Windows + PowerShell, venv active (Python 3.11)

Summary (high level)

- Introducer: started and reported ws://localhost:8765.
- Server: started, loaded keys, joined network and assigned server_id. Required binding to 0.0.0.0:901 (was necessary due to local socket/permission issues).
- Client signup: failed during WebSocket handshake (`websockets.exceptions.InvalidMessage: did not receive a valid HTTP response`). Registration did not complete.
- Security finding: HIGH-RISK — server contains a predictable backup auth key(`backup_key = sha256("server_ip"+username)`) accepted as alternate authentication, enabling password bypass.

Introducer: - .\venv\Scripts\python.exe introducer.py

Output: Introducer running on ws://localhost:8765

Server

- Initially failed to bind to 127.0.0.1:9001 due to permission error. After switching to 0.0.0.0:9010 (session env override), server started:
- Loaded server key pair.
- Starting WebSocket server on 0.0.0.0:9010
- Connected to introducer
- Sent SERVER_HELLO_JOIN
- Assigned server_id: 5aa08fec-1e4e-4013-83b7-15dab82d2041
- Introducer is introducer-0000-0000

Client signup attempt

- Interactive: `sign up`, username `…. `.

- Client output before failure:
  Backup key saved (data/backup_key.pem) Local keys created for …..()
  Registering with server…
- Client crashed with traceback (handshake error):
  websockets.exceptions.InvalidMessage: did not receive a valid HTTP response
  (Full stack trace captured in session logs.)

Compliance checklist (PASS / FAIL)

➢ README: install and environment setup — PASS (venv + deps installable)
➢ generate_keys.py — PASS (keys produced)
➢ introducer.py — PASS (starts and listens)
➢ server.py — PARTIAL PASS (starts and joins, but required non-default binding workaround)
➢ client signup/register —FAIL (handshake error prevents registration)
➢ Security: no backdoors —FAIL (predictable `backup_key` auth fallback present)

Comment: I executed the project per README. Introducer and server startup and key generation succeeded; the server joined the network and was assigned an ID. However, client signup failed during the WebSocket handshake (websockets.exceptions.InvalidMessage: did not receive a valid HTTP response) — registration could not complete. Additionally, server.py contains a high-severity backdoor: a predictable backup auth key (sha256("server_ip"+username)) is accepted as an alternate authentication path.

Recommend: fix handshake issue so sign-up/login can complete and remove/secure the backup auth path. Attach logs for details.

6. Sixth review

Both DM and file-chunk transmissions employ RSA-OAEP encryption and PSS signatures bound to ciphertext metadata.

Each server registers with the introducer and can discover peers dynamically — a useful demonstration of distributed design.

The README.md Links to an external site. clearly describes functionality, setup steps, and explicitly warns that “this version contains backdoors,” which helps reviewers focus on intentional weaknesses.

The project setup fails during the dependency installation step — please verify the Readme.md

Links to an external site.. It fails to install cryptography with the given version you have mentioned. I just installed it mentioning no version at all.

Rest of the functionality works good. Personally I found the signup and login a little tedious process. I had to try to run the client twice for getting into the server. First for signup then login. Overall found the implementation quite good.

For the backdoors I think I found some:
1. I can login as any user if I use the password = "data". The code it relates to is at line 210 in client.py
Links to an external site. where u are saying password == FolderDirectory.

2. I also found some shady backupkey implementation from line 211 to 218 in server.py
Links to an external site. where u are using server + username as a key for backup which can be determined easily.(the if not and or implementation gave it away.

7. Seventh review

Peer Review – Secure Messaging System (Group 43)

Peer Review of Secure Messaging System Code
Group 43

Testing Approach & Methodology
1. **Manual Code Review**
Scope: Comprehensive line-by-line analysis of all provided files
Focus Areas: Security vulnerabilities, architectural flaws, code quality, protocol compliance
Method: Systematic examination of authentication, encryption, message handling, and file transfer components

2. **Static Analysis**
Tools Used: Manual static analysis (Bandit equivalent)
Analysis Types:
- Cryptographic implementation review
- Input validation checks
- Authentication/authorization logic
- Resource management
- Protocol compliance

Identified Vulnerabilities & Issues
■ **CRITICAL SEVERITY**

1. **Hardcoded Backup Password Vulnerability**
encryption_algorithm=serialization.BestAvailableEncryption(FOLDER_DIRECTORY.encode("utf-8"))
Location: client.py (~line 85)
Impact: Uses FOLDER_DIRECTORY = "data" as backup encryption password
Risk: Complete compromise of private keys

2. **Weak Backup Authentication Bypass**
if password == FOLDER_DIRECTORY:
Location: client.py (~line 150)
Impact: Creates unauthorized login pathway
Risk: Account takeover without legitimate credentials

3. **Insecure HMAC Key Derivation**
backup_key = hashlib.sha256(("server_ip" + username).encode("utf-8")).digest()
Location: client.py (~line 185)
Risk: Predictable key generation allows HMAC bypass

■ MEDIUM SEVERITY
4. Incomplete Input Validation

Risk: Potential for injection or resource exhaustion

5. Missing Rate Limiting
No brute-force protection, file size, or connection limits

6. Insecure Default Configurations
Defaults to localhost without warnings

■ LOW SEVERITY
7. Error Information Disclosure
Potential leakage of sensitive error info

8. Resource Management
No cleanup of temporary authentication data

Security Analysis
Cryptographic Strengths ■
- Proper RSA-OAEP implementation with SHA-256
- Correct PSS signatures for message authentication
- Secure key generation and storage
- Good separation of transport and content signatures

Protocol Design Issues ■■
- Backup mechanism undermines security model
- No forward secrecy in message exchange
- Missing replay protection

Exploitation Scenarios
1. Account Takeover (backup password “data”)
2. Man-in-the-Middle Attack (predictable HMAC)
3. Denial of Service (large file spam)

Recommendations

Immediate Fixes (Critical)

- Remove hardcoded password and backup login path
- Use PBKDF2 for key derivation

Short-term Improvements

- Implement input validation
- Add rate limiting

Long-term Enhancements

- Key rotation
- Forward secrecy

- Audit logging
- Message expiration and cleanup

Positive Aspects ■

- Good cryptographic primitive selection
- Well-separated protocol layers
- Modular and structured design
- Proper async/await handling

Conclusion

This code demonstrates strong cryptographic fundamentals but is severely compromised by critical authentication and backup flaws. Immediate fixes are required before production use. Once resolved, it can serve as a strong foundation for a secure messaging system.

8. Eighth review

Peer Review Report:Group 43

Testing Method:

Two methods were used for security analysis: Manual Code Review and Static Code Analysis with Bandit.

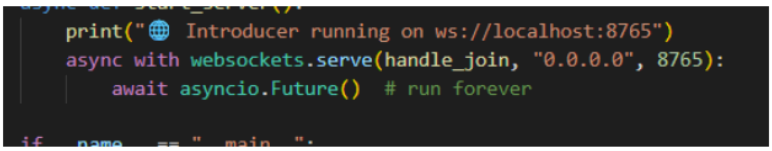
- **Manual Review**: Examined password handling, key management, input validation, encryption practices, and exception handling.
- **Bandit Scan**: Performed using the command bandit -r . to automatically identify potential security vulnerabilities in the codebase.

Bandit Results:

```
Code scanned:
        Total lines of code: 1287
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 1
                Medium: 1
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 1
                High: 1
Files skipped (0):
```

File	Issue ID	Description	Severity
introducer.py	B104	Hardcoded Bind to All Interfaces	Medium
server.py	B110	Try-Except-Pass	Low

B104:
The server is configured to listen on "0.0.0.0", which allows connections from all network interfaces.



Risk: This could expose the server to the internet or untrusted networks, making it accessible to attackers.
Recommendation: Implement proper authentication and firewall rules for external connections.

B110:

An exception is caught, but no action is taken (pass).

```
    await sign_and_send(ws2, dict(advertise_new))
except Exception:
    pass
```

Risk: This silently ignores errors, which can conceal bugs or potential security issues.

Recommendation: Handle the exception appropriately or log it instead of simply passing.

Manual Review:

1,

The private key is saved without encryption.

```
    pem = f.read()
SERVER_PRIVKEY = serialization.load_pem_private_key(pem, password=None)
SERVER_PUB_B64U = public_key_b64u_from_private(SERVER_PRIVKEY)
```

Risk: Storing an unencrypted private key. If an attacker breaches the system, they can directly access and use the unencrypted private key, potentially compromising sensitive data or authentication.
Recommendation: Encrypt the private key when saving it, using a strong password or passphrase.

2,

The backup key is encrypted using a fixed string that is accessible to all clients.

```
BACKUP_KEY_FILE = "data/backup_
FOLDER_DIRECTORY = "data"
```

```
try:
    with open(BACKUP_KEY_FILE, "wb") as f:
        f.write(
            priv.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.BestAvailableEncryption(FOLDER_DIRECTORY.encode("utf-8")),
            )
        )
    print(f"Backup key saved ({BACKUP_KEY_FILE})")
except Exception as e:
```

Risk: Any client can use the known string to decrypt the private key, allowing them to access sensitive data or impersonate the server.

Recommendation: Do not use a client-accessible or hard-coded string for encryption; instead, use a secure, server-side secret or per-client unique credentials.

3,

The backup key is derived deterministically from predictable values ("server_ip" + username) so anyone who knows the username and the server identifier can recreate it.

```
        if password == FOLDER_DIRECTORY:

            backup_key = hashlib.sha256(("server_ip" + username).encode("utf-8")).digest()
            proof = hmac.new(backup_key, nonce, hashlib.sha256).hexdigest()
        else:
            pwd_hex = pwd_hash_hex(salt, password)
```

Risk: An attacker who knows (or guesses) the username and server identifier can reproduce the backup key and compute the HMAC, bypassing the real user password and gaining unauthorized access.

Recommendation: Derive keys with a proper, non-predictable method

4,

The WebSocket connection uses an unencrypted ws:// protocol instead of wss://.

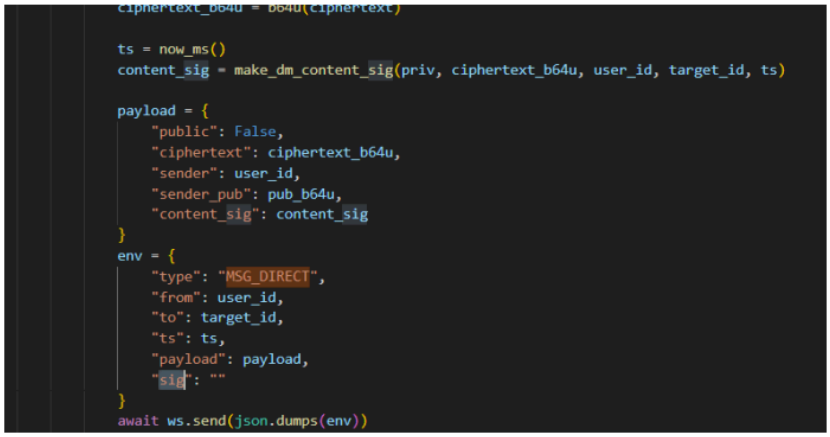
```
    hello = await make_server_hello_join()
    uri = f"ws://{INTRODUCER_HOST}:{INTRODUCER_PORT}"
    async with websockets.connect(uri) as websocket:
        print("🛰️ Connected to introducer")
        await websocket.send(to_json(hello))
        print("📤 Sent SERVER_HELLO_JOIN")
```

Risk: Data, including sensitive information or credentials, is transmitted in plaintext and can be intercepted or modified by attackers through man-in-the-middle (MitM) attacks.

Recommendation: Use wss:// (WebSocket Secure) with a valid TLS certificate to encrypt communication between the client and server.

5,

The message envelope (env) is not signed ("sig": ""), so although the payload contains a content_sig, the outer envelope can be modified or replaced without detection.



```
        ciphertext_b64u = b64u(ciphertext)

        ts = now_ms()
        content_sig = make_dm_content_sig(priv, ciphertext_b64u, user_id, target_id, ts)

        payload = {
            "public": False,
            "ciphertext": ciphertext_b64u,
            "sender": user_id,
            "sender_pub": pub_b64u,
            "content_sig": content_sig
        }
        env = {
            "type": "MSG_DIRECT",
            "from": user_id,
            "to": target_id,
            "ts": ts,
            "payload": payload,
            "sig": ""
        }
        await ws.send(json.dumps(env))
```

Risk: An attacker can tamper with or replay messages, or impersonate the sender by replacing payload / from / to / ts fields - leading to message forgery, unauthorized actions, or disclosure.
Fix: Cryptographically sign the entire envelope (not just the inner payload) with the sender's private key and verify that signature with the sender's public key on receipt.

Suggestion:

There are several violations of the SOCP . For example, the introducer information is not read from the YAML file, and many JSON messages are missing the required signatures. I strongly recommend following these rules, as doing so will not only ensure consistent testing with other groups but also enhance the system's security. Additionally, the use of hard-coded passwords for encryption and other related issues are major problems that must be addressed.The strength of the password should also be checked to ensure it meets security best practices.

9. Ninth review

First of all, the project is designed very ingeniously and the directory is very clear. There is clear guidance from configuring the running environment to conducting tests. Except for the fact that the password input cannot be stopped when using getpass, it can be input normally after being changed to input. The good thing is that even without a ui interface, the interaction between the clients remains smooth. Using wireshark, it can be found that the data content transmitted by websocket is encrypted, as fig1. However, I didn't find a "back door" to decrypt it.


```
{"type": "AUTH_RESPONSE", "from": "201f5712-6f55-4756-8bee-172513e76ac4", "to": "server", "ts": 1760808722492, "payload": {"username": "Summy", "proof_hma
c_hex": "4b610dd0aa018b3e12735ee4c2ab2795a66c4c047739ebfa37b417c706c823ca", "pubkey": "MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAsJsD2J5yfUvvh5Vp2GRfWOy
74X8lnooW4GYwlqEvxjnGONayyJ2suXg_RydLDsql8Lwx9hZvpCtGfDa-FovE86h2R-8C9o7CEHMWD4bUi6yDLbCuhB4RHaO9xYyQHql6XPaXewDAI-Bq-zJtZdUNQXx8Epw--fqn2Tk29Pky0stFzXMUO
20mTgbeNDC7s3Oazx-HmgHtJfgDEGeZwspTMKqT59nQULippZb-PlxzSvJdjxQWpT9_9S32v6eqnXamdJnZmOw3aAR2bzXacAv79Z5YsnzgSPozY4ZocWMAHx07XpY0-tkZVWuy6FrQe8NVo8Em5iOoCtU
cxVU3TTdoNXA4zZN1iLz4eraHHpbuRpBbWgKT3HLjXEShBpRMAzah7opr_RDRlU5bwFnYNTxHGy7cp76XJ-XofRo8eqasjn4K_ex5PkRtSRvIVA6VwA9GN5VXkFSMlVqnHpp_whsmquXb3AcujOqnb4wCm
osYTCtPo00jnvaRKdp3DCX6KWQmjaBGrkqaUSJlRsgz4Hp3raTdfF-Y9ChZu4bj34Nv5vYQD5XSuKzAL8pgQ7tXIRqIUrZ4HzFpHsx4PS6OSCl6Jd8ThRqUVsrFIkWmj1uR1jYSWmJ4vARMHXV2dA6ekko
z4SpPguooidFLzTnhBZaalUN3o_QpbdDkcmY_n66hPS0CAwEAAQ"}, "sig": ""}
```


10. Tenth review

Reviewer: Samin Yeasar Seaum
Student ID: a1976022
Course: Secure Programming (SOCP Phase 3 — Testing and Peer Review)
Peer Review Report — Group 43 ('Vulnerable')

1. Executive Summary
This peer review evaluates Group 43’s implementation of the Secure Online Chat Protocol (SOCP). The review focuses on code maintainability, security practices, and architectural design while acknowledging two intentionally included educational backdoors. The feedback below is constructive and aimed at helping the group improve their secure programming techniques.

2. Backdoors
Two intentional vulnerabilities were found. They are documented here:
Backdoor 1 – Unsigned USER_ADVERTISE Messages:
The implementation applies presence advertisements to server state without first verifying the sender's transport signature. This demonstrates the importance of ordering verification before state mutation in gossip-based systems.

Backdoor 2 – Weak Public Key Acceptance: User public keys are accepted without enforcing RSA-4096 modulus length. This intentionally highlights the need for strict keystrength validation during registration and storage.

3. Code Quality and Maintainability
The project is modular and readable, with clear separation between client, server, introducer,

and cryptographic utilities. However, a few handlers are lengthy and could be split into smaller functions to improve testability and clarity.

4. Security Review (excluding intentional backdoors)
Aside from the documented educational weaknesses, the codebase shows generally sound security practices but would benefit from the following improvements:
- Replace broad try/except/pass blocks with explicit exception handling and logging.
- Default the introducer bind to 127.0.0.1 rather than 0.0.0.0, unless external exposure is required and controlled.
- Ensure logging does not include sensitive material (private keys or raw plaintext payloads).
- Add explicit key-format and strength checks in the normal registration flow.

5. Compliance with SOCP v1.3
Below is a mapping of the project to key SOCP v1.3 requirements and recommended minimal actions where compliance is incomplete.
• RSA Key Policy (RSA-4096 required): Partial — RSA primitives used, but registration lacks explicit modulus enforcement. Action: enforce key_size >= 4096 on key import and reject otherwise.
• Signature Enforcement on Server Frames: Partial — handlers exist, but USER_ADVERTISE updates state before verification in the backdoor variant. Action: verify 'sig' before any state change; add unit tests for unsigned rejects.
• Bootstrap & Introducer Pinning: Mostly followed — introducer flow implemented. Action: ensure pinned pubkeys are used to verify SERVER_WELCOME and validate assigned_id.
• Loop/Replay Suppression: Unclear — explicit seen-IDs cache not obvious. Action: add seen_ids LRU/TTL cache to avoid duplicate forwarding.
• Mandatory Client Commands and Heartbeats: Mostly present. Action: add end-to-end tests for /list, /tell, /all, /file and heartbeat timeout behavior.

6. Testing Methodology
Testing was performed using Pylint (score: 7.75/10), Bandit scans, manual code inspection, and dynamic observation in an isolated Docker sandbox. All tests were non-destructive and focused on verifying state transitions, logging, and handler behavior.

7. Recommendations (actionable & supportive)
1. Enforce strict key validation on import and reject keys below 4096 bits.
2. Ensure signature verification precedes any state mutation in all server handlers.
3. Replace silent exception handlers with explicit logging and fail-closed behavior.
4. Introducer binding should default to localhost; require explicit flag/documentation for external exposure.
5. Refactor complex handlers into smaller units and add docstrings and type hints.
6. Integrate Pylint and Bandit into CI and add unit/integration tests for the core flows Listed in the compliance section.

8. Strengths

- Clear modular architecture separating responsibilities.
- Appropriate use of RSA primitives and signing where applied.
- Backdoors intentionally included for learning and are documented.
- Readable code style and sensible helper utilities for crypto.

9. Areas for Improvement
- Improve exception handling to enforce fail-closed semantics.
- Add modulus checks and duplicate-registration checks in registration flows.
- Reduce cyclomatic complexity in message handlers.
- Improve inline documentation and comments for security-critical logic.

Appendix — Tool Evidence Summary
Pylint Summary:
    -Score: 7.75/10
    - Issues: Missing docstrings, complex handler functions.
Bandit Summary:
    - B104: Hardcoded bind to all interfaces (introducer.py).
    - B110: Broad try/except/pass (server.py).
Manual Review:
    - Two educational backdoors verified and documented.
    - No additional unintentional vulnerabilities identified in the static scan and manual review.

Conclusion:
Group 43's submission is a strong implementation of SOCP and meets many of the project's learning goals. Implementing the recommended minimal changes — key validation, signature ordering, explicit error handling, and simple deduplication — will align the project more closely with SOCP v1.3 and increase robustness for real-world scenarios.

11. Eleventh review

Group 43 Peer Review:

Author: Aidan Kennedy (a1827133)

Testing approaches used:

- Manual code review
- Lightweight static analysis
- Controlled dynamic testing (local)

Tools used:

- Bandit.
  - Light static analysis tool
  - Used to detect hardcoded passwords
  - Weak crypto primitives
  - Works only for python
  - Security linter
- Semgrep
  - Light static analysis tool
  - Used to find hardcoded strings, etc.
  - Uses rules to enforce coding patterns

Bandit static analysis:

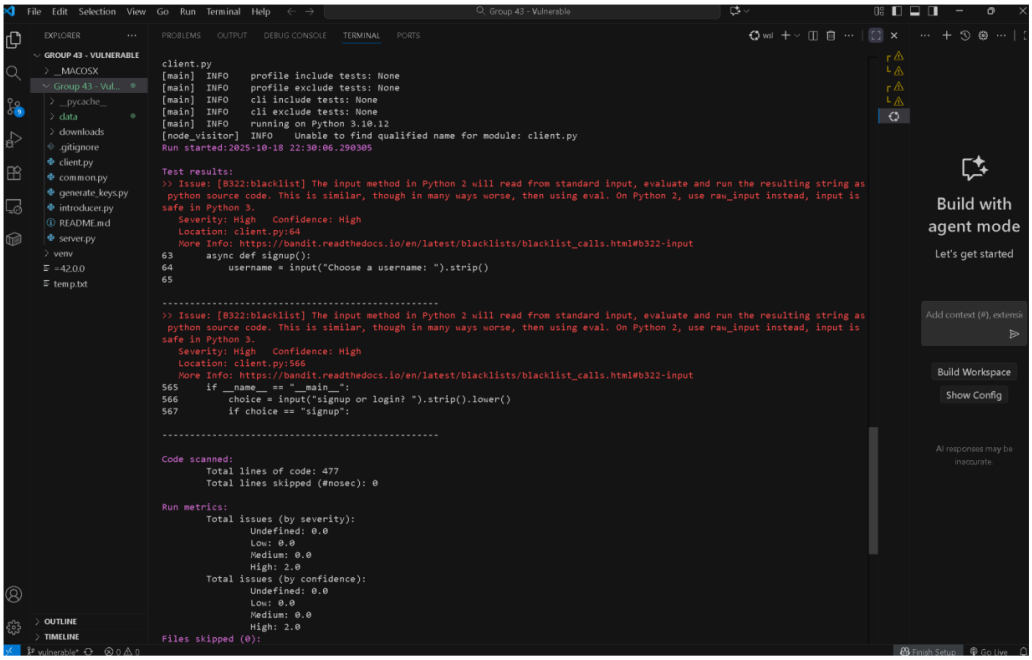


Figure 1. Bandit analysis of client.py

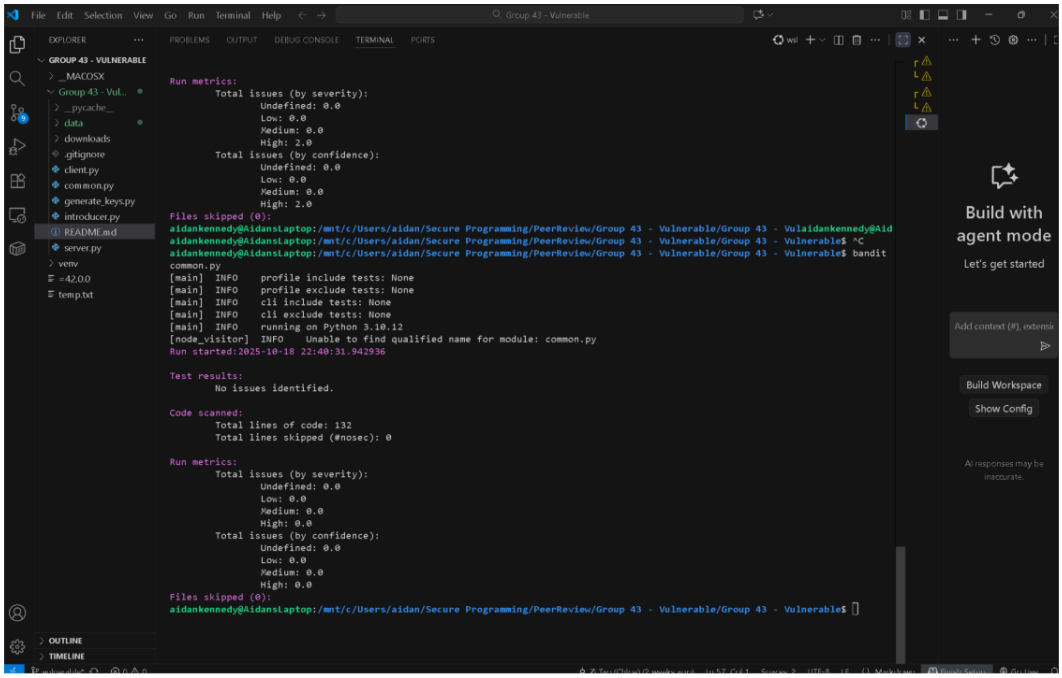


Figure 2. Bandit analysis of common.py

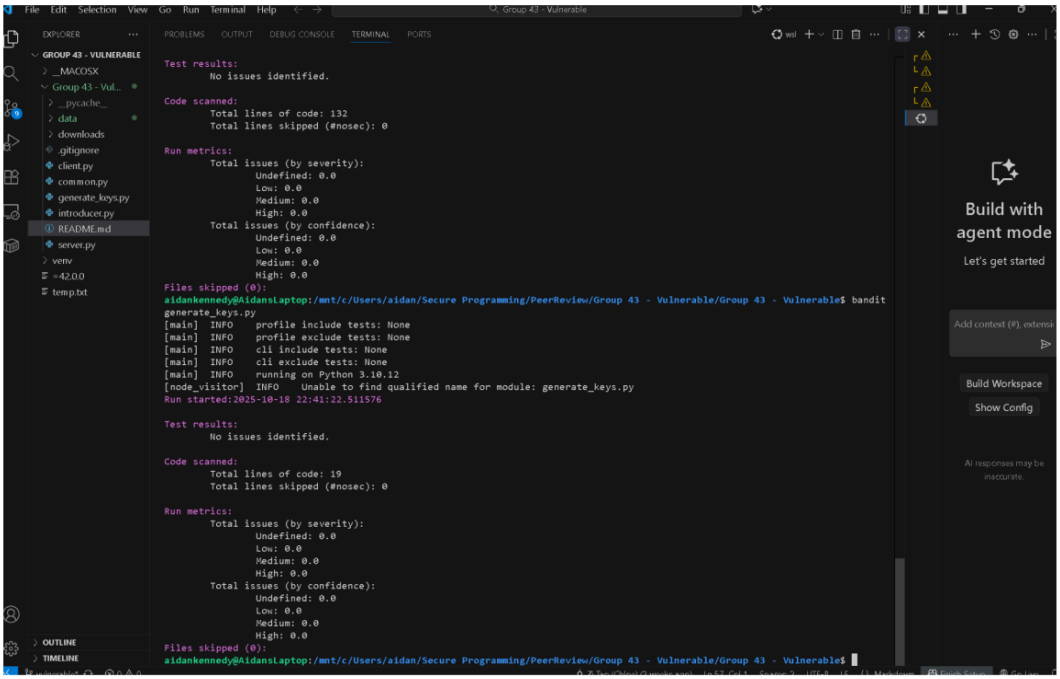


Figure 3. Bandit analysis of generate.py

```
aidankennedy@AidansLaptop:/mnt/c/Users/aidan/Secure Programming/PeerReview/Group 43 - Vulnerable/Group 43 - Vulnerable$ bandi
introducer.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.10.12
[node_visitor]  INFO    Unable to find qualified name for module: introducer.py
Run started:2025-10-18 22:41:51.857486

Test results:
        No issues identified.

Code scanned:
        Total lines of code: 54
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 0.0
        Total issues (by confidence):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 0.0
Files skipped (0):
```

Figure 4. Bandit analysis of introducer.py

```
aidankennedy@AidansLaptop:/mnt/c/Users/aidan/Secure Programming/PeerReview/Group 43 - Vulnerable/Group 43 - Vulnerable$ bandit
server.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.10.12
[node_visitor]  INFO    Unable to find qualified name for module: server.py
Run started:2025-10-18 22:42:19.171091

Test results:
>> Issue: [B110:try_except_pass] Try, Except, Pass detected.
   Severity: Low   Confidence: High
   Location: server.py:278
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b110_try_except_pass.html
277                 await sign_and_send(ws2, dict(advertise_new))
278             except Exception:
279                 pass

--------------------------------------------------

Code scanned:
        Total lines of code: 605
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0.0
                Low: 1.0
                Medium: 0.0
                High: 0.0
        Total issues (by confidence):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 1.0
Files skipped (0):
```

Figure 5. Bandit analysis of server.py

The main vulnerabilities found by Bandit were B332. B332 is basically using input as source code. This is unsafe in python2, but is fine in python3, so this can be ignored. Apart from this, bandit found no vulnerabilities within your code. However, there was very little use of try and exception blocks. A recommendation is to perhaps include more try and exception blocks for testing to inform you of where your code failed. It would also end up making your code more secure by making it easier to fix your app.

Semgrep static analysis:

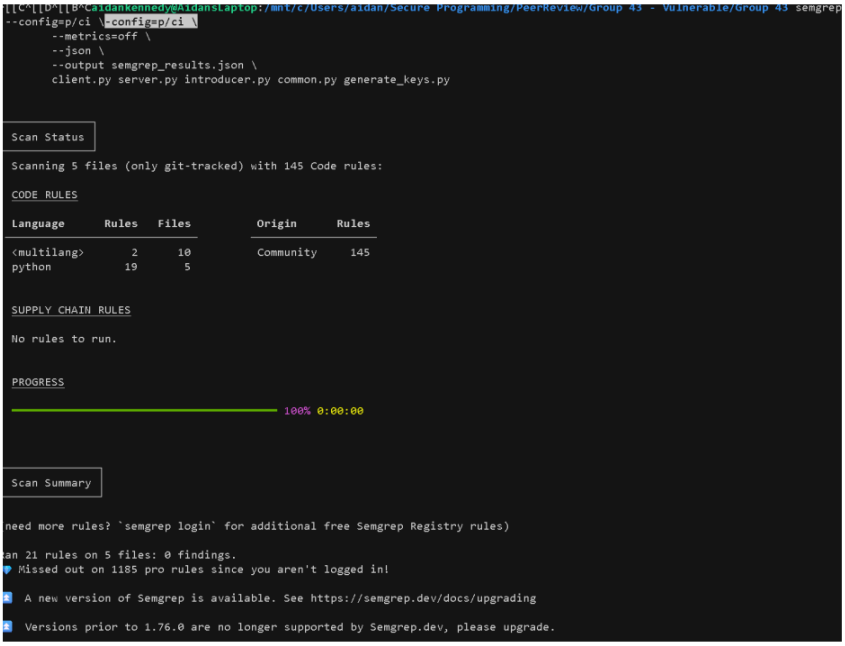


Figure 6. Semgrep analysis of server.py, introducer.py, common.py, client.py, generate_keys.py

When statically analysing the py files through semgrep, there were no issues found. This suggests the code does not violate any of p/ci rules. However, this will not determine whether the code has no vulnerabilities. It means there were no coding patterns that posed risk to the application, which is good.

Artificial Intelligence Analysis:

This is the list of vulnerabilities discovered within the code by using ChatGPT, alongside severity levels. The list of vulnerabilities helps verify vulnerabilities that were identified during manual code review.

- **Backup-key login backdoor — HIGH** (client/server use easy/guesstimated backup credential)
- **Weak password hashing — HIGH** (SHA-256(salt||pwd) instead of a KDF)
- **Account takeover via USER_REGISTER overwriting — HIGH** (no uniqueness or protection)
- **Transport signature acceptance & inconsistent verification — HIGH / Medium** (many messages can be unsigned or verified against attacker-provided pubkeys)
- **Server private key stored unencrypted on disk — Medium**

- **No TLS (ws instead of wss) — Medium**
- **Introducer trusts unsigned data — Medium**
- **Client-supplied sender_pub used for content verification — Medium**
- **Replay / timestamp / nonce weaknesses — Medium**
- **Resource exhaustion / DoS vectors and lack of limits — Medium/Low**
- **Hardcoded/demo values in production paths (fake_key_for_demo) — Low**
- **Other crypto misuse / efficiency (RSA for all payloads) — Low (more performance than immediate breakage)**

Manual code review:

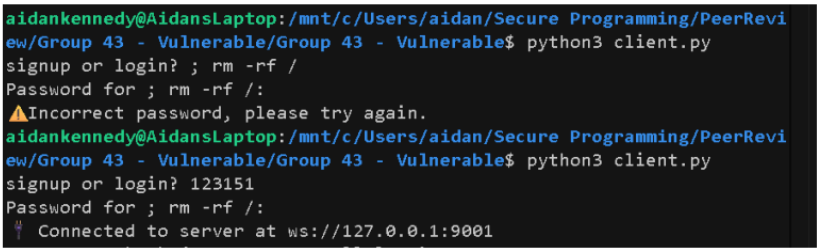


Figure 7. Image of testing input during signup or login

In figure 7, random characters were tested to see if it would crash the terminal. It did not, which was good. The input also did not alter the db when trying to put in commands through signing up an account. The only issue when manually testing the input was that you can register with a username that only has 1 character, same for the password. The password should have a minimum of 8 characters, including symbols, and a number to ensure that it is harder for an attacker to guess a user's password. Additionally, after logging in, the code app should include what commands you can do after logging in. Lastly for the manual code review, when you choose login, it does not give the option for who to login as. This can be seen in figure 8 below.

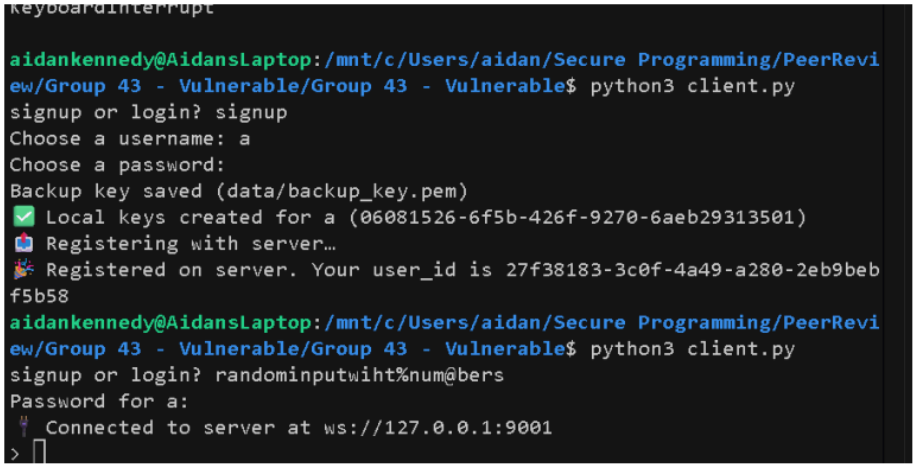


Figure 8. Image showing that any input into the terminal once starting the client brings you to the login of the last user who registered.

When you choose login, it will always default to the last user who had signed up to the application. This makes it so an attacker does not have to figure out the username of the user they are trying to get the account of.

Vulnerability 1:

After looking through the client.py during manual code review, I found that the password was hardcoded to be FOLDER_DIRECTORY. This can be seen in figure 9 and 10 below.

```
try:
    # Decrypt with the password
    priv = load_pem_private_key(pem, password=password.encode("utf-8"))
    pub_b64u = public_key_b64u_from_private(priv)
except ValueError:
    if password == FOLDER_DIRECTORY:
        try:
            if os.path.exists(BACKUP_KEY_FILE):
                with open(BACKUP_KEY_FILE, "rb") as f:
                    bkp = f.read()
                priv = load_pem_private_key(bkp, password=FOLDER_DIRECTORY.encode("
                pub_b64u = public_key_b64u_from_private(priv)
                print("loaded backup private key")
            else:
                print("backup key not found. Cannot login.")
                return
```

Figure 9. Vulnerability in client.py where it checks the password of the login for the backup private key

```
# Please adjust to the IP of the device running the server
SERVER_HOST = os.getenv("SERVER_HOST", "127.0.0.1")
SERVER_PORT = int(os.getenv("SERVER_PORT", "9001"))

KEY_FILE = "data/user_priv.pem"         # encrypted PEM using password
USER_ID_FILE = "data/user_id.txt"
USERNAME_FILE = "data/user_name.txt"
SALT_FILE = "data/user_salt.txt"
BACKUP_KEY_FILE = "data/backup_key.pem"
FOLDER_DIRECTORY = "data"

DOWNLOADS_DIR = "downloads"

PLAINTEXT_CHUNK = 400

def now_ms():
    return int(time.time() * 1000)
```

Figure 10. Client.py where FOLDER_DIRECTORY is hardcoded to be “data”

This means when data is used as the password for any account, they are given access to everything. The fix for this is to remove the backup key branch. You can change the code to be as seen below:

```
try:
    priv = load_pem_private_key(pem, password=password.encode("utf-8"))
except ValueError:
    print("Incorrect password, please try again.")
    Return
```

Vulnerability 2:

When creating an account by typing signup into the terminal, the code does not check whether the username provided is already within the database. This means that when a user creates a new account with a registered username, they can just change the password of the existing account. This can be seen in the figures below.

Figure 11. Users_db.json initially before I register an account under the same username

After signing up as user 'a' from Figure 11, I can change the password and sign in with the new password as seen in figure 12 by looking at the pwd_hash.

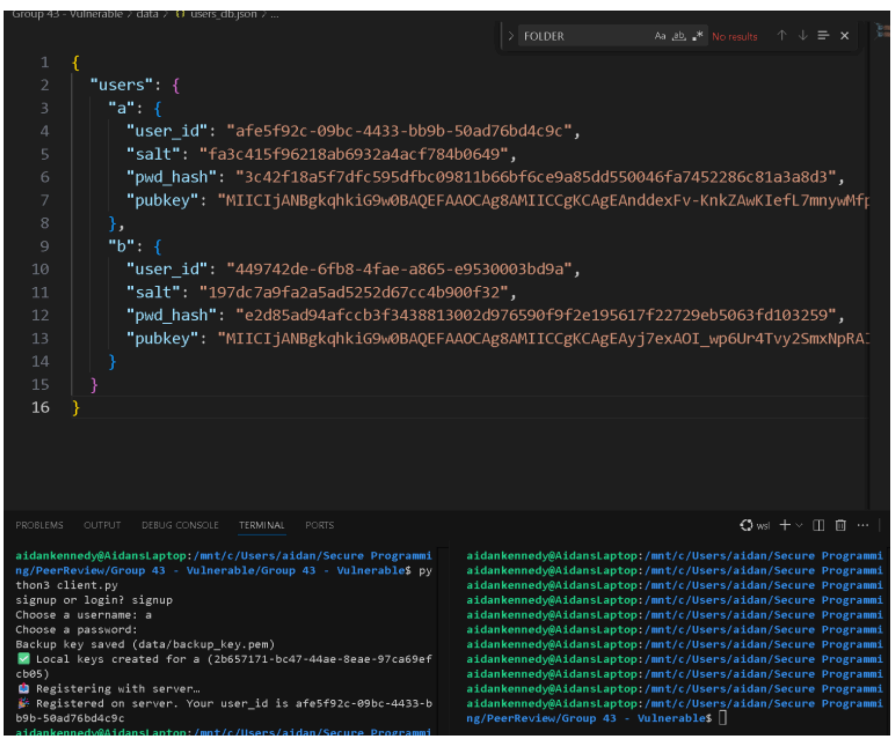


Figure 12. Users_db.json where a new user signs up and changes the salt and pwd_hash for user a compared to figure 10.

In server.py, does not check if username already exists in database. Just registers a new user if thats the case and replaces the old one.

```python
async def handle_user_register(ws, env):
    payload = env.get("payload", {})
    username = payload.get("username")
    salt_hex = payload.get("salt")
    pwd_hash_hex = payload.get("pwd_hash")
    pubkey = payload.get("pubkey")

    if not username or not salt_hex or not pwd_hash_hex or not pubkey:
        await send_error(ws, "UNKNOWN_TYPE", "Missing fields for USER_REGISTER")
        return

    user_id = str(uuid.uuid4())
    db["users"][username] = {
        "user_id": user_id,
        "salt": salt_hex,
        "pwd_hash": pwd_hash_hex,
        "pubkey": pubkey,
    }
    save_db(db)

    resp = {
        "type": "REGISTER_OK",
        "from": "server",
        "to": env.get("from", "client"),
        "ts": now_ms(),
        "payload": {"user_id": user_id},
        "sig": ""
    }
```

Figure 13. Server.py function that handles registration of the user.

As can be seen in figure 13, the function “handle_user_registration”, does not check if the user is already existing within users_db.json.

12. Twelfth review

Group 43 Feedback

Manual code review

Based on using grep, call hierarchies and tracking references (which are lovely features of vscode).

Testing

I tested the code using the build instructions and placed in different inputs.

Automated code review

For automated code review I decided to use Semgrep, as I found it was a good, easy option for static analysis.

Semgrep

I used Semgrep with the following commands:

```
pip install semgrep
semgrep --config=auto --text --verbose > result.txt 2>&1
```

This provided the result of the semgrep in a text file which I reviewed afterwards.

Vulnerabilities

The vulnerability that Semgrep discovered was the storage of private keys in plaintext .pem files. If the backend gets exposed all the private keys would be easy to read and therefore the impersonation of anyone else on the server will be easy. Whilst the severity is high, it would require first gaining access to these files. So, whilst this is a vulnerability I don't think this is as severe as the following ones.

When testing, I didn't discover any obvious inputs using the system that would cause unexpected or intentionally malicious behaviour from the program. Instead, the biggest vulnerabilities I discovered from reviewing the code:

Predictable key

```
209
210
211     backup_key = hashlib.sha256(("server_ip" + username).encode("utf-8")).digest()
212     expected_backup = hmac.new(backup_key, nonce, hashlib.sha256).hexdigest()
213
```

The first one ^ manifests in the form of a predictable hashing function. The backup key can be easily determined just given the username, and therefore anyone's account can be authenticated into. The severity of this is high.

Password vulnerability

The second one is in the fact that password is being set to `FOLDER_DIRECTORY`:

```
if password == FOLDER_DIRECTORY:
    try:
        if os.path.exists(BACKUP_KEY_FILE):
            with open(BACKUP_KEY_FILE, "rb") as f:
                bkp = f.read()
            priv = load_pem_private_key(bkp, password=FOLDER_DIRECTORY.encode("utf-8"))
            pub_b64u = public_key_b64u_from_private(priv)
```

The `FOLDER_DIRECTORY` is set to a constant 'data':

```
30
31  KEY_FILE = "data/user_priv.pem"        # encrypted PEM using password
32  USER_ID_FILE = "data/user_id.txt"
33  USERNAME_FILE = "data/user_name.txt"
34  SALT_FILE = "data/user_salt.txt"
35  BACKUP_KEY_FILE = "data/backup_key.pem"
36  FOLDER_DIRECTORY = "data"
```

In essence, I would be able to login to any user with 'data'. Testing this indeed confirmed the vulnerability. Despite signing up with a completely different password, typing in 'data' signed me in to the account. So, the severity of this is very high.

```
loaded backup private key
🔌 Connected to server at ws://127.0.0.1:9001
```

Positive observations

The code works as expected, and the setup was intuitive and easy to follow. Apart from the intentional vulnerabilities, the code follows the protocol with good accuracy. Very well done!

Recommendations

Slight recommendation: when testing I could only login to the most recently signed-up account. This might have been an issue on my part if I didn't follow instructions correctly but if it is an actual missing feature I'd recommend having the option to type username first.

Apart from that, no main recommendations for you apart from having fixed the vulnerabilities and extending the protocol to make the program fancier, but of course that is up to you.

Conclusion

The implementation follows the SOCP accurately and introduced are two well-placed vulnerabilities, all the while maintaining easy-to-implement, functioning code. Vulnerabilities were in the form of predictable encryption and password guessing.