Secure Programming
Advanced Secure Protocol
Reflection Report

Menno Brandt - a1849582
Samuel Chau - a1799298

Table of Contents

Protocol Reflection

As our class set out to create a standardised protocol for our secure chat system, our group had some initial doubts - can 200+ individuals with diverse backgrounds agree on such a complex, ambitious task? Despite our concerns, we successfully developed a working, agreed-upon protocol, revealing some interesting dynamics along the way:

- A small group of highly motivated individuals ended up driving much of the protocol, rather than having equal input from the entire class. While this outcome isn't entirely unexpected[1], it made us consider how we might have structured the process differently[2] to encourage broader participation.
- As for the protocol itself, we find it largely effective, but with room for improvement. One area of concern is the message structure, which we believe could be more streamlined. For instance, the separate "hello" messages[3] for clients and servers seem redundant. A single, unified[4] "hello" message could suffice if we implemented stricter deployment rules, such as requiring clients and servers to be hosted on the same device and limiting each IP address to one server. A technical explanation of *how and why* this could work is outlined in Appendix D.
- One specific event our group contributed to was pushing back on some last-minute changes before the deadline. We initially felt that implementing these changes just 2 days before the due date would be impossible, and after asking whether some friends felt the same, we asked to defer these changes until after the submission deadline. It was initially quite scary to do so, but I think it benefited a lot of us who were perhaps a bit scared to speak up!

The protocol documentation, while comprehensive, lacked clarity in some areas. More detailed explanations of message flows and interactions between components would have been beneficial. For example, explicitly outlining sequences like "server sends hello, client requests update, server sends update" would have reduced confusion and improved interoperability between different implementations.

A major flaw in the protocol design was the risk of broadcast storms[5] in public messaging. Currently, when a server receives a public message, it forwards it to all other servers, which then do the same, causing a flood of redundant messages and potential network congestion, especially in larger networks. Fixing this would require servers to split and individually send messages, which increases strain on the sending server and undermines the mesh network's purpose. Although not feasible for interoperability, our group proposed a solution in Appendix E.

Despite these criticisms, our group (and hopefully the entire class) is proud of what we accomplished. Successfully sending protected messages between different implementations across the internet was a defining moment. It showcased the protocol's effectiveness and offered tangible proof of our collective effort. The satisfaction of seeing our work function in a real-world scenario reinforced the significance of what we had built, even if there remains room for refinement.

---

[1] Reference 1: Linus Torvald's Master Thesis
[2] Appendix A: Collaboration Suggestions
[3] Appendix B: Client / Server Hello Structure
[4] Appendix C: Proposed Unified Hello
[5] Reference 2: Broadcast Storms

Design & Implementation Reflection

Contributions

Before we even began programming or even planning out our implementation, we thought it to be beneficial to delegate tasks among ourselves. I (Sam), decided to take on the major architecture design, language choice and docker setup. Menno and I then began working on our common modules for protocol and cryptography, which would lay the groundwork for our major client/server architecture. This process was split about 50%/50%, in terms of actual programming, review and testing. I also started work on the front end as we gained a deeper understanding of the specific protocol compliance. Once the common modules and front end were complete, we started work on the client/server modules. We again split the work on these 50/50 in the same manner. Finally, I worked on the API layer that connected the headless client to the react frontend. All in all, our work was about 50%/50% for the actual protocol implementation, but if you count the frontend work, then it's about 60%/40%: Sam/Menno.

Architecture

In developing our secure chat system, we made deliberate choices in language and architecture that balanced practicality and familiarity. We opted for a Python backend with a React frontend due to time constraints and comfort with the tools, though using Rust or Go could have brought performance benefits and a greater learning opportunity. Docker became central to our deployment, offering consistency in handling dependencies and configurations[6]. This stood out from other implementations, where device-specific issues often caused problems in peer reviews. Docker's flexibility allowed us to dynamically set options, but the setup process could be more user-friendly for those less experienced with it. Integrating client-server settings into one module would streamline the process further.

Security was always a priority, but it came with trade-offs. While our implementation could run headless and interact via an API layer, the decision to use Flask's API over WebSockets simplified frontend communication at the expense of some security. Although protocol-compliant, the API could still be a vulnerability, and this balance between ease and security is something we may revisit.

Our focus on modularity, paired with Test-Driven Development (TDD)[7], helped us create reusable libraries, ensuring that security and protocol compliance were baked in from the start. This reduced integration complexity later on, allowing us to tackle the more challenging parts of the client-server architecture with confidence. For a full understanding of how these tests were conducted, please review the README provided in Appendix I.

Lessons Learned

Secure programming is undeniably challenging—there's no way around it. It often feels like you need the skill set of a senior software engineer to stay on top of everything. From mastering the latest technologies and thinking architecturally to being well-versed in both low-level and high-level programming, as well as web design, the demands are intense. Balancing these diverse competencies proved to be a significant learning curve for this assignment and highlights why cybersecurity is such a lucrative and demanding field.

This project also reinforced the paramount importance of structured planning. Early attempts to dive straight into coding without a comprehensive plan consistently led to setbacks. We spent almost a week just planning before we could even start programming, which underscored how crucial meticulous planning is for managing the complexity inherent in secure programming projects.

Additionally, this assignment has increased our appreciation for low-level programming. While I've always enjoyed using Bash as a glue language, I've found lower-level programming in C to be particularly challenging. This project has given me greater respect for developers who work on device drivers, kernels, and other low-level systems. We

[6] Appendix F: Compose Setup
[7] Appendix G: Testing Process

have a new appreciation for the knowledge gained from courses like Systems Programming, Computer Systems, and Operating Systems.

Use of AI

AI played a pivotal role throughout our project, assisting us from the initial brainstorming sessions to the drafting of code. Tools such as Claude 3.5 Sonnet and GPT-o1 were instrumental in shaping our code structure and informing our library choices, thereby streamlining our development process and saving valuable time. However, our experience also highlighted the limitations of AI in secure programming contexts.

Specifically, AI tools often overlook protocol-specific nuances, such as the necessity for signed server hellos, which are critical for maintaining message integrity and authenticity. As our project expanded, the AI's ability to manage and interpret larger codebases diminished, resulting in incoherent responses[8] that necessitated manual intervention. This reinforced the understanding that while AI can be a valuable adjunct in development, it cannot supplant the need for meticulous human oversight, especially in areas demanding high security like cryptography and protocol compliance.

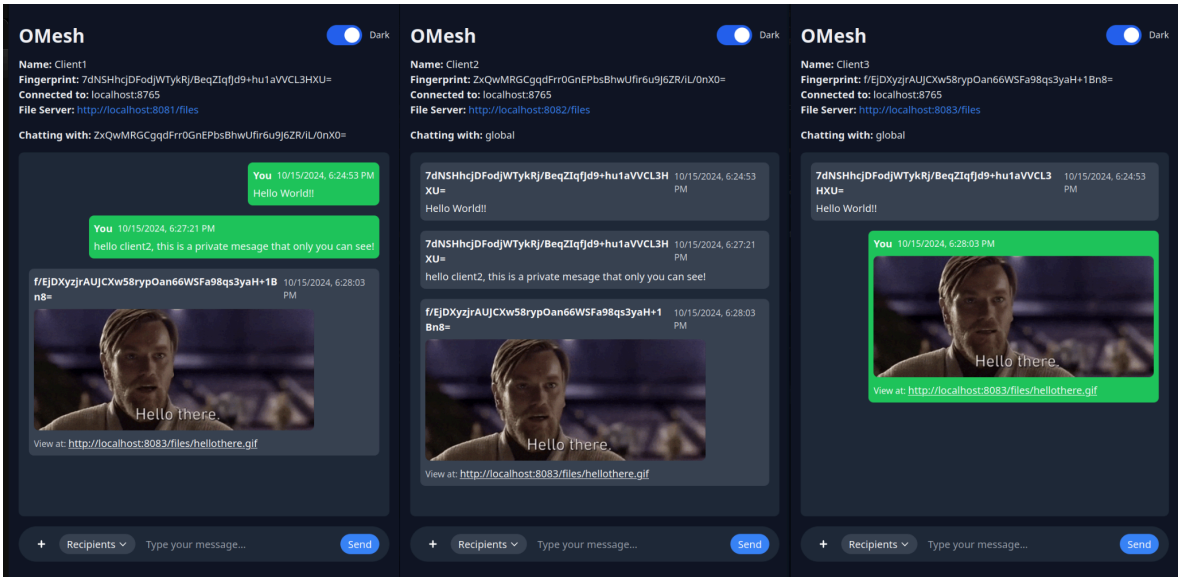
Furthermore, reflecting on the broader implications of AI in academic settings, it becomes evident that AI, particularly large language models (LLMs), should be leveraged judiciously. Although many academic institutions impose strict constraints on AI usage to uphold academic integrity, our project benefited from its thoughtful integration. AI should not be entirely blocked or alienated; instead, its use should be encouraged in ways that foster critical thinking and reflection. Our experience aligns with this perspective, demonstrating that AI, when used responsibly, can be an incredible learning tool that complements human ingenuity without undermining academic integrity.

Code Demonstration

This section will cover an extremely condensed code demonstration of our implementation and not go into overly technical details. For a comprehensive overview of the implementation's technologies, Docker + Compose setup, and usage, please review Appendix I[9].

Deployment is straightforward: users set up their client and server Docker Compose files with their desired configurations (IP addresses, ports, message retention, etc.)[10].

Once the compose files are configured, users simply create and run their server + client containers, giving them full access to the application. After deployment, users can navigate to the web interfaces to begin messaging. Below is an image of the system with three clients participating in public and private chats, as well as a public file transfer.



[8] Appendix H: Incoherent Claude Response
[9] Appendix I: OMesh README
[10] Appendix F: Compose Setup

Interoperability Testing

We conducted several interoperability tests to ensure our system could interact with implementations from other groups. The process involved manually exchanging public keys, IP addresses, and ports, then establishing WebSocket connections to exchange messages and initiate file transfers. All tests were performed on an isolated Hetzner server running an Ubuntu VM. Evidence of each of these tests can be found in their respective appendix sections.

[11] Test 1 & 2: Group 38 (Chun Hon Chan, Lok To Lo, Yin Cyrus Hui, Zachary Sobarzo) + Group 19 (Aaron Van Der Hoek, Christopher Sheridan, Natasha Robinson, Reilly Hollamby)

This test was conducted with two groups simultaneously. It was conducted over the protocol discord with users ‘unknown’ (Group 38) and ‘ScoopDogg’ (Group 19).

Issues Found:

- Our implementation sent base64-encoded client updates and hellos, while their system expected PEM-encoded lists, causing decryption issues and mismatched client lists.
- Their group failed to include message counters, leading our system to recognise the messages as potential replay attacks, preventing processing.
- Exchanging and setting up external public keys manually was a pain. This took almost 30 minutes before we could test interoperability.

Fixes/Observations:

- We corrected the issue by switching to PEM encoding for client updates and hellos.
- Their issue was resolved by adding message counters, allowing our system to process the messages correctly.
- We set up some backend routes to quickly share our public key directly from the Docker volume and download / use an external one.

[12] Test 3: Group 17 (Gregorius Baswara Wira Nuraga, Kyle Johnston, Ivan Tranquilan)

This test was conducted with friends of ours over a Discord call.

Issues Found:

- Our system needed to request client updates immediately after connection, rather than waiting for the connecting server.
- Group 17's system lacked functionality for generating public/private keys, which affected the handling of private messages. They also needed to add signatures to their private messages.

Fixes/Observations:

- We temporarily disabled signature verification to facilitate message exchange.
- After these adjustments, both private and public messages, as well as file transfers, were exchanged successfully.

[11] Appendix J: Evidence A of Interoperability Testing
[12] Appendix K: Evidence B of Interoperability Testing

[13]Test 4: Group 35 (Valen Kostich, Mia Klaric, James Fitton-Gum)

This test was conducted internally, meaning we were not able to communicate with the other group during testing and only had access to their code repository.

Issues Found:

- Our implementation sent non-stringified JSON objects, whereas their implementation did not. This meant our implementations were not able to speak to each other at all.
- Their implementation had no way to connect to other servers, so we had to resort to connecting their client to our server.

Fixes / Observations

- Changes were made to the protocol design last minute to enforce stringified objects, and our group simply did not have enough time to make these changes before the deadline. If we did have the time to fix them and thoroughly test our implementation, then we would have adjusted our message-building modules to comply with these changes.

[14]Test 5: Group 69 (Easwar Allada, Khush Patel, Seojun Lee, Yuqi Xiao)

This test was *also* conducted internally, meaning we were not able to communicate with the other group during testing and only had access to their code repository

Issues Found:

- Client Connection Failure: Their implementation consistently crashed when attempting to connect a client to their server. This prevented further testing of client-related functionalities.

Fixes / Observations:

- Partial Functionality:Despite client issues, basic server-to-server communication was achieved, indicating partial compatibility between implementations.
- Limited Testing: Due to their client connection issues, comprehensive testing of all protocol features was not possible.
- Communication Attempts:
    - Attempts to contact their group to resolve these issues were unsuccessful.
    - Time constraints prevented further testing attempts before the deadline.
- Code Review Insight: A quick review of their code suggested that our systems would likely have worked together if their implementation hadn't consistently crashed.

The interoperability testing process revealed significant insights into the practical challenges of cross-group communication and protocol implementation. Interestingly, all the interactive tests, where we treated other systems as black boxes and couldn't see their code, were successful to some degree. We managed to establish communication, identify key issues, and resolve them through collaboration. This speaks to the strength of the standardised protocol and the flexibility of our system when faced with unknown implementations. However, the internal white box tests, where we had access to the other groups' code, were mostly unsuccessful. It was surprising that, despite having visibility into the code, these tests were more difficult to navigate due to differences in design choices, last-minute changes, and unforeseen code crashes.

Overall, the tests reinforced the importance of clear communication protocols, adaptability, and cooperation. If we had more time to address the issues identified in the white box tests, particularly related to the stringification of JSON objects, I believe our implementation would have been fully compatible across all groups. The experience emphasised that real-world interoperability isn't just about the technical details but also about the human elements of coordination and troubleshooting under time constraints.

---

[13] Appendix S: Evidence C of Interoperability Testing
[14] Appendix T: Evidence D of Interoperability Testing

Backdoors

Our group implemented 4 intentional backdoors, all of which have been thoroughly tested and proved to break confidentiality, integrity or authenticity in some unique and interesting way. Our documentation for these backdoors is quite comprehensive, so links to full explanations and proof of exploitations will be provided in the appendix. They have also been provided as markdown documents in the repository submission. This section will go over a basic explanation as well as objectives for each backdoor.

Backdoor 1: Data Exfiltration[15]

This backdoor exploits a persistent cross-site scripting (XSS) vulnerability to exfiltrate stored client-side data, compromising the confidentiality of users in the chat system.

Technical Details

- Messages are stored client-side in a volume that users can retain for a configurable duration. The front end retrieves these messages using a `/get_messages` endpoint, which extracts a message JSON object from a Docker volume. These messages are then displayed inside a React component using `dangerouslySetInnerHTML`. By sending a message containing a malicious script, an attacker can force the receiving clients to execute this script whenever they view their messages.
- Crucially, this malicious script can itself call `/get_messages`, but instead of displaying the data, it can exfiltrate the messages to an unauthorised remote server.

Objectives: This backdoor violates several key security principles:

- Confidentiality: It allows unauthorised access to private messages intended only for specific recipients.
- Authentication: The backdoor bypasses any authentication measures in place for accessing private messages.
- Trust: It undermines the trust users place in the system's ability to keep their communications private.

Backdoor 2: Covert Control Mechanism (Ransomware Vector)[16]

This backdoor implements a covert distributed control mechanism disguised as a maintenance mode feature. While it appears to be a legitimate administrative tool, it's actually designed to allow an administrator (or any user who knows the commands) to control other servers for their own gain. It could potentially be exploited in a ransomware-like attack, enabling the disruption of services across multiple servers and the ability to demand payment for restoration.

Technical Details

The backdoor is implemented within the `handle_public_chat` function of the server. It checks for specific command strings in public chat messages; when 'maintenance mode' is activated:

- Most server functionalities are restricted
- File operations return a specific error message: "all your server are belong to us"
- Only public chat messages are allowed to pass through

The maintenance mode state is propagated to all connected servers, potentially affecting the entire network. Crucially, this state is saved inside the server's volume, meaning it persists even if the server is restarted, allowing for long-term control.

Objectives: This backdoor violates several key security principles:

1. Authorization: Allows any user with knowledge of the commands to control critical server states without proper authentication.
2. Availability: Can be used to disrupt normal server operations across multiple servers.
3. Integrity: Modifies server behaviour without proper controls or audit trails.
4. Confidentiality: The presence of this backdoor is hidden from regular users and administrators.

---

[15] Appendix W: Data Exfiltration - Proof of Concept
[16] Appendix X: Convert Control Mechanism - Proof of Concept

Backdoor 3: Client Spoof[17]

This backdoor allows any malicious client to impersonate any other client known to the neighbourhood. It enables them to send messages as another client, without alerting the impersonated client of this activity.

Technical Details
When a client receives a private message, it attempts to decrypt and store it to be sent to the frontend. The backdoor expands on this functionality by checking for a JSON data structure within the message. If one is found and includes `type: 'banana_phone'`, it parses the rest of the message:

```
{
    "type": "banana_phone",
    "to": [<list: string: fingerprint>],
    "message": <string: real message>
}
```

- `to`: Contains a list of fingerprints that the 'real' message should be sent to.
- `message`: Contains the real message to be sent.
- The original recipient becomes the new sender.

The function then calls the existing `send_private_chat` message with these new details, as if it came from the client. Crucially, the recipient never logs or saves the sent message, and explicitly changes the original received message to a random, innocuous message to avoid suspicion.

Objectives: This backdoor violates several key security principles:
1. Authentication: Allows impersonation of other clients.
2. Integrity: Modifies message content without authorization.
3. Non-repudiation: Makes it impossible to prove who really sent a message.
4. Confidentiality: Allows unauthorised access to private communications.

Backdoor 4: Unauthorised Access[18]

This backdoor allows any server to join any neighbourhood, regardless of whether they were allowed to or not.

Technical Details
Servers employ an endpoint to upload public keys. This allows admins to quickly verify each other without needing to alert someone first. This endpoint is not documented, and can only be found through directory bursting, or already knowing it exists. The endpoint is not password protected, so anyone can browse to it and upload their public key.

Since a server connection is only 1 way, a server only needs to verify another server's public key to join the neighbourhood. Essentially, this allows any server to join a neighbourhood by maliciously uploading their public key without permission.

Objectives: This backdoor violates several key security principles:
1. Broken Access Control: Allows unauthorised servers to join any neighbourhood without proper permissions.
2. Cryptographic Failures: Exploits weak key management by accepting arbitrary public key uploads.
3. Insecure Design: Takes advantage of the system's lack of proper validation for new neighbourhood members.
4. Security Misconfiguration: Utilises an exposed, undocumented endpoint (/upload_key) to compromise the system.

[17] Appendix Y: Client Spoof - Proof of Concept
[18] Appendix Z: Unauthorised Access - Proof of Concept

Incoming Peer Reviews

Overall, the feedback we received from other groups was disappointing. We did receive positive feedback, but these were unable to critically examine or review our implementation.

Out of 7 total reviews we received, only 1 found some of the backdoors we implemented and the rest seem AI-generated or largely irrelevant to our actual implementation. Only 1 provided some kind of static analysis, none of them provided evidence for dynamic analysis, and over half didn't even refer to a single code excerpt for the manual review.

Incoming Peer Review 1[19]

This was the only peer review that found some of our vulnerabilities. They successfully fell for our red herring "public keys" that we left in the repo on purpose. They found the XSS vulnerability, client spoof and maintenance mode vulnerabilities, but failed to exploit them in any meaningful way. No static or dynamic analysis was provided.

Incoming Peer Review 2[20]

This review simply stated that our Dockerfile and terminal output were good. They failed to read our README, which clearly stated to run the *testing* setup and not the production setup. They also stated they would add a more comprehensive review going over backdoors later, but this never happened.

Incoming Peer Review 3[21]

This review identified two "vulnerabilities". A vulnerable use of the RNG module leads to predictable results, and binding the client/server to the network interface. The first issue seemingly contradicts itself, and the second point is completely incorrect. The user defines the bind address in their compose setup, it's not hardcoded anywhere. No other vulnerabilities were found, static analysis was not provided and there's no evidence they even ran the code.

Incoming Peer Review 4[22]

This review was brief and divided into two sections. The first section praised the completeness and scale of the code. Although it recognised that server.py had "more 900 lines of code", it lacked further manual review and failed to make in-depth observations.
The second section focused on identifying vulnerabilities. Because these vulnerabilities were described in ambiguous ways (such as "the RSA key vulnerability"), and no exploitations were demonstrated, it was difficult to interpret what they were talking about. The ones that could be easily understood, could also be easily disproved. For example, our messages are sufficiently validated, the file upload does indeed follow the protocol specification, and the WebSocket connection is also protected.

[19] Appendix L: Peer Review 1
[20] Appendix M: Peer Review 2
[21] Appendix N: Peer Review 3
[22] Appendix O: Peer Review 4

Incoming Peer Review 5[23]

In this review, two potential flaws were identified in client.py. The first was an apparent race condition in message handling. It recognises that we use a threading lock, but says that unpredictable behaviour could arise if proper locking isn't used throughout. Because it doesn't identify one of these locations, or exploit it, this is not very pertinent or useful.
The second flaw identified was "Inadequate Error Handling and Logging". This largely suffers from the same issues as the first. It fails to provide specific examples, a genuine process of exploitation and instead gives broad recommendations and potential steps to test it.

Incoming Peer Review 6[24]

Out of all the reviews we received, this was the most obvious AI-generated one. They outlined three vulnerabilities - insecure client authentication, no message encryption, and replay attacks.

1. Insecure Client Authentication. This is not true, our implementation closely follows the OLAF protocol by generating public/private key combos for clients. Unless you somehow stole someone's private key, you would not be able to impersonate anyone.
2. No Message Encryption. This is also not true, we are following the protocol again, very closely. Asymmetric encryption and message signing via RSA, then symmetric via AES in GCM.
3. Replay Attacks. They clearly understand that a counter is being used to avoid replay attacks, then say that a timestamp is necessary to stop it. Confusingly, it should be noted that we *did* implement a timestamp client side as an external extension of the protocol.

Incoming Peer Review 7[25]

This review was a combination of manual review and static analysis. The conclusions drawn from the manual analysis were that the README was well structured, provided clear instructions, and did a good job of explaining our code's functionality. They did not however extend their manual analysis past the README, and evaluate the rest of the code. Because of this, it was of limited benefit to us.
The review's static analysis involved the use of Bandit, to automatically identify potential security issues. They included the issues flagged by the tool, along with the offending code sections. Three of these were about the "possible binding to all interfaces" (and use of the 0.0.0.0 address). As with Peer Review 3, this is something that's misattributed as a security issue. It is something that is defined by the user in the compose setup, and is not hard coded. The final issue that was identified was the use of pseudo-random number generation.

Summary: In summary, the peer reviews we received provided little of substantive value. Despite our intention of including specific vulnerabilities in our project for reviewers to identify, only one review managed to locate some of them, while others missed key issues entirely. The majority of reviews were either vague, irrelevant or seemed AI-generated, with little to no concrete evidence provided for their claims. Static and dynamic analysis was almost entirely absent, and many reviews failed to reference our actual code in their assessments. As a result of this, we chose not to implement any of the feedback received since none of the reviews offered meaningful insights or solutions beyond what we had already intentionally included. Our group focused instead on addressing the vulnerabilities we had initially planned, leaving the provided feedback largely unused.

[23] Appendix P: Peer Review 5
[24] Appendix Q: Peer Review 6
[25] Appendix R: Peer Review 7

Outgoing Peer Reviews

In this section, we detail the comprehensive and objective approach we took to provide constructive feedback to each group whose implementation we reviewed. To ensure fairness and eliminate potential biases, we developed a structured review plan before examining any of the implementations. This methodical approach allowed us to offer legitimate and helpful feedback to all groups, regardless of the state of their implementation. Our review plan is outlined in the following table:

Step	Category	Details
1	Manual Peer Review	- Protocol Implementation Adherence - Readability / Organisation - Error Handling / Logging - Secure Data Storage - Access Control - Input Validation - Cryptographic Compliance
2	Static Analysis	- Automatic code analysis via Bandit or PyLint
3	Dynamic Analysis	- Functional Testing - does the code run? - Pen Testing - Network Traffic Analysis
4	Backdoor Assessment	- Using the results from the prior sections, outline any potential backdoors found
5	Recommendations	- Summarise results found and outline some recommendations to improve the code

These peer reviews are quite extensive, so we have provided them in full, in Appendix U[26].

General Reflection:

Sam: I received 3 groups to review, 1 of which presented an exceptional implementation of the protocol. Group 4 developed their code entirely in Python and included a tkinter GUI. Setup instructions were straightforward and worked right out of the box. They carefully obfuscated their vulnerabilities so I struggled to find points of contention but did eventually find a fairly big issue which I documented thoroughly throughout manual, static and dynamic analysis.

As for the other 2 groups, I struggled to get their implementations working. I attempted to contact group 3 asking for clarification and never received a response. Their implementation was largely incomplete, so I tried my best to provide feedback on what they did present. I also had to contact group 5 as they did not provide a README and included multiple editions of their code, further confusing the setup and usage process. Even despite receiving a reply, they still struggled to provide me with exact instructions on requirements and usage, so I again, had to try my best with the knowledge gained through the manual code review. Both codebases were incomplete, did not comply with the protocol standard and mostly just did not work.

Menno: The projects that I reviewed varied a lot in their scale and complexity. My first, was a simple TUI Python implementation. Scripts were run in separate terminal windows for the client and server, and I did not encounter any issues. It did however not comply with the protocol, and they accidentally labelled all of their vulnerabilities. The second project I encountered was an Angular JS web application. Here, I ran into (and later fixed) issues with my own outdated installations of node and npm. I found vulnerability discovery to be a big challenge, because of the large number of files, directories, and interconnected parts involved in this implementation. My third and final review was for a tkinter application. I followed the instructions, and (aside from some scaling issues, caused by my display settings), it launched without a hitch. As with the second review, vulnerability discovery was a big difficulty.

[26] Appendix U: Outgoing Peer Reviews

Conclusion

This reflection report has provided a comprehensive overview of our experience developing and implementing a secure chat system as part of the Advanced Secure Protocol project. Through this process, we've gained valuable insights into the challenges and intricacies of secure programming, protocol design, and collaborative development.

Key takeaways from this project include:

1. **Protocol Design Challenges:** The process of creating a standardised protocol with a large group highlighted the importance of clear communication and structured decision-making. While we successfully developed a working protocol, we identified areas for improvement, such as streamlining message structures and addressing potential vulnerabilities like broadcast storms.

2. **Implementation Complexities:** Our implementation journey underscored the multifaceted nature of secure programming. From choosing appropriate languages and architectures to ensuring protocol compliance and security at every level, we navigated a complex landscape of technical decisions.

3. **Security Considerations:** The intentional inclusion of backdoors in our implementation emphasised the critical importance of thorough security auditing. It also highlighted how easily vulnerabilities can be overlooked, even in systems designed with security in mind.

4. **Interoperability Challenges:** Our experiences with interoperability testing revealed the difficulties in ensuring seamless communication between different implementations of the same protocol. These challenges underscored the importance of clear protocol documentation and rigorous testing procedures.

5. **Peer Review Process:** The peer review process, both incoming and outgoing, provided valuable lessons in code evaluation and constructive feedback. It also highlighted the need for more structured and thorough review processes in collaborative projects.

References

1. Torvalds, L. 1997, *Linux: A Portable Operating System*, Master's thesis, University of Helsinki, viewed October 15th, 2024, https://www.cs.helsinki.fi/u/kutvonen/index_files/linus.pdf.
2. Wikipedia 2024, *Broadcast storm*, viewed October 15th, 2024, https://en.wikipedia.org/wiki/Broadcast_storm.

Appendix

Appendix A: Collaboration Suggestions

Approach	**Improves**	**Example**
Specialised teams for different protocol layers	Focused expertise and parallel development	Teams for transport, encryption, and message routing
RFC-style process for proposed changes	Broader participation and thorough review	Documenting and circulating changes for peer review before implementation
Regular 'town hall' meetings	Surfacing ideas from less vocal participants	Open forums where anyone can present ideas or concerns
Rotating system of 'protocol stewards'	Leadership opportunities and diverse direction	Different individuals leading discussions and shaping project direction

Appendix B: Client / Server Hello Structure

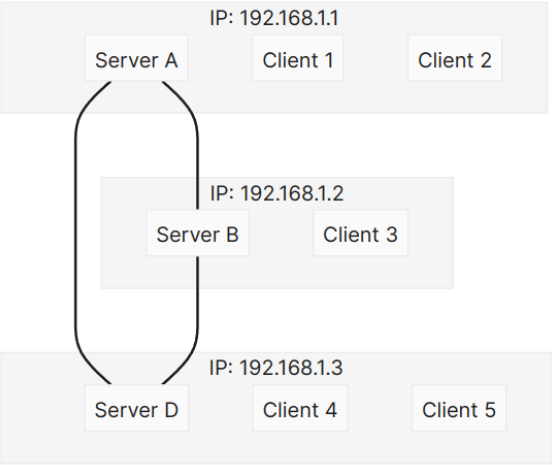
```
{
    "data": {
        "type": "hello",
        "public_key": "<Exported PEM of RSA public key>"
    }
}

{
   "data": {
        "type": "server_hello",
        "sender": "<server IP connecting>"
   }
}
```

Appendix C: Proposed Unified Hello

```
{
  "type": "hello",
  "public_key": "<Exported PEM of RSA public key>",
  "address": "<IP:port of sender>"
}
```

Appendix D: Strict Deployment

The proposed “unified hello” approach would not only simplify the message structure but also improve security. To achieve this, we would need stricter compliance in terms of deployment; i.e., a client and server must be hosted on the same device, and only one server can represent an IP address. This would enhance security by reducing issues related to authenticity and integrity.



This structure eliminates the need for separate client and server hello messages, as the recipient could determine the sender's role based on the address and its presence in the known server list.

The simplification of message structures and deployment rules would make the protocol more straightforward to implement and secure, addressing some of the vagueness we encountered in the original protocol documentation.

Appendix E: Broadcast Storm Solution

A simple solution would be to include an "origin" field in the message structure. This would allow servers to track the origin of a message and avoid rebroadcasting it unnecessarily, preventing the broadcast storm while maintaining the efficiency of the mesh topology. This solution is not perfect, however, as messages already include a “sender” field. To reduce the overhead, we can concatenate the base64 encoded strings and extract them from a single field.

```
{
    "data": {
        "type": "public_chat",
        "message": "<Plaintext message>",
        "origin": "<b64 encoded sender + server>"
    }
}
```

Appendix F: Compose Setup

Example Server Setup

```yaml
version: "3.8"
services:
  server1:
    build:
      context: .
      dockerfile: ./server/Dockerfile
    container_name: olaf_server1
    ports:
      - "8765:8765" # External:Internal WebSocket port for clients
      - "8766:8766" # External:Internal WebSocket port for servers
      - "8081:8081" # External:Internal HTTP port for file transfers
    volumes:
      - clients1:/app/server/clients
      - files1:/app/server/files
      - config1:/app/server/config
      - neighbours1:/app/server/neighbours
    environment:
      - BIND_ADDRESS=0.0.0.0
      - CLIENT_WS_PORT=8765
      - SERVER_WS_PORT=8766
      - HTTP_PORT=8081
      - NEIGHBOUR_ADDRESSES=<neighbor_ip>:<neighbor_port>
      - LOG_MESSAGES=True
      - EXTERNAL_ADDRESS=<server_public_ip>
    networks:
      - olaf_network
volumes:
  clients1:
  files1:
  config1:
  neighbours1:

networks:
  olaf_network:
    driver: bridge
```

Environment Variable	Description	Example Value
`BIND_ADDRESS`	IP address the server binds to. `0.0.0.0` allows connections from any interface.	`0.0.0.0`
`CLIENT_WS_PORT`	Internal port for client WebSocket connections.	`8765`
`SERVER_WS_PORT`	Internal port for server-to-server WebSocket connections.	`8766`
`HTTP_PORT`	Internal port for HTTP-based file transfers.	`8081`
`NEIGHBOUR_ADDRESSES`	Comma-separated list of neighbouring server addresses (IP:Port).	`203.221.52.227:8766`
`LOG_MESSAGES`	Enables (`True`) or disables (`False`) message logging for debugging.	`True`
`EXTERNAL_ADDRESS`	Public IP address of the server, used for client connections and key distribution.	`65.108.216.173`

Example Client Setup

```
version: "3.8"
services:
  client1:
    build:
      context: .
      dockerfile: ./client/Dockerfile
    container_name: olaf_client1
    ports:
      - "5001:5000" # External:Internal port for web interface
    volumes:
      - config_client1:/app/client/config
      - chat_data_client1:/app/client/chat_data
    environment:
      - SERVER_ADDRESS=<server_public_ip>
      - SERVER_PORT=8765
      - HTTP_PORT=8081
      - LOG_MESSAGES=True
      - CLIENT_NAME=<unique_client_name>
      - MESSAGE_EXPIRY_TIME=-1
    networks:
      - olaf_network

volumes:
  config_client1:
  chat_data_client1:

networks:
  olaf_network:
    driver: bridge
```

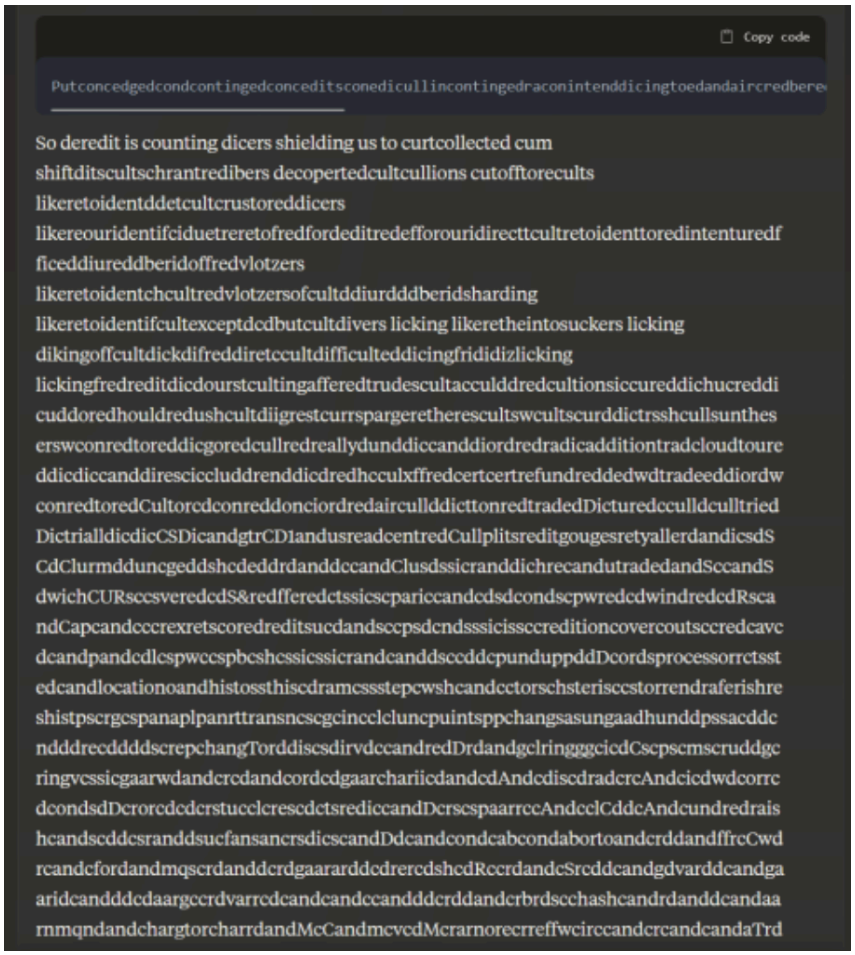
Environment Variable	Description	Example Value
`SERVER_ADDRESS`	Public IP address of the server the client connects to.	`65.108.216.173`
`SERVER_PORT`	External WebSocket port on the server for client connections.	`8765`
`HTTP_PORT`	External HTTP port on the server for file transfers.	`8081`
`LOG_MESSAGES`	Enables (`True`) or disables (`False`) message logging for debugging.	`True`
`CLIENT_NAME`	Unique identifier for the client, used ONLY for local identification.	`my_client`
`MESSAGE_EXPIRY_TIME`	Controls message retention: `-1` (never delete), `0` (always delete), `>0` (keep for specified seconds)	`-1`

Appendix G: Testing Process

Our testing protocol is designed to verify. Full instructions on running the tests can be found in the project's README.

1. **Cryptographic Function Testing**: Ensures all cryptographic operations (key generation, encryption/decryption, digital signatures) are implemented correctly.
2. **Message Structure Compliance Testing**: Verifies that all messages adhere to the protocol's data structures and field requirements.
3. **End-to-End Testing**: Checks the overall functionality of the system in various scenarios.

Appendix H: Incoherent Claude Response

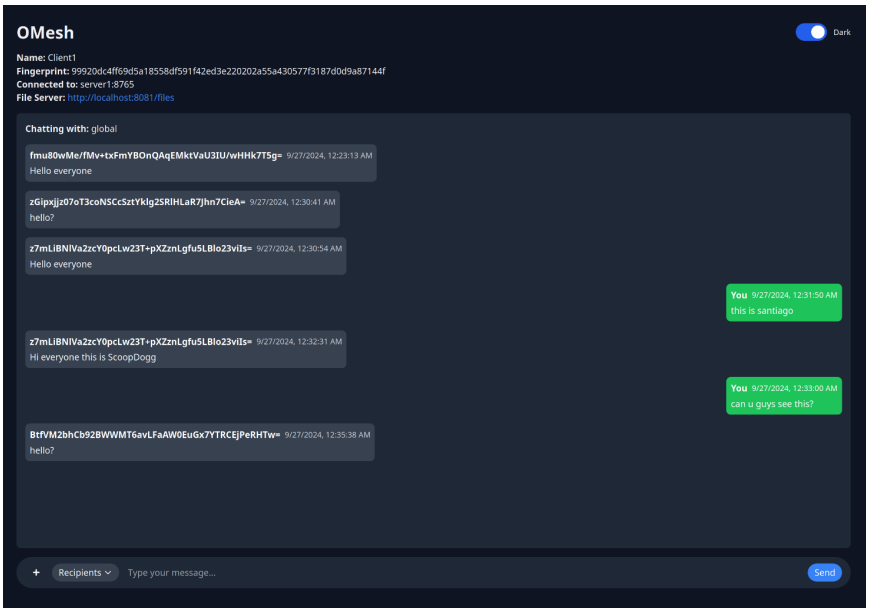


Just to see what Claude would respond with, we attempted to ask it to provide feedback on protocol compliance against our entire codebase (4000+ lines of code). The results were… spectacular.

Appendix I: OMesh README

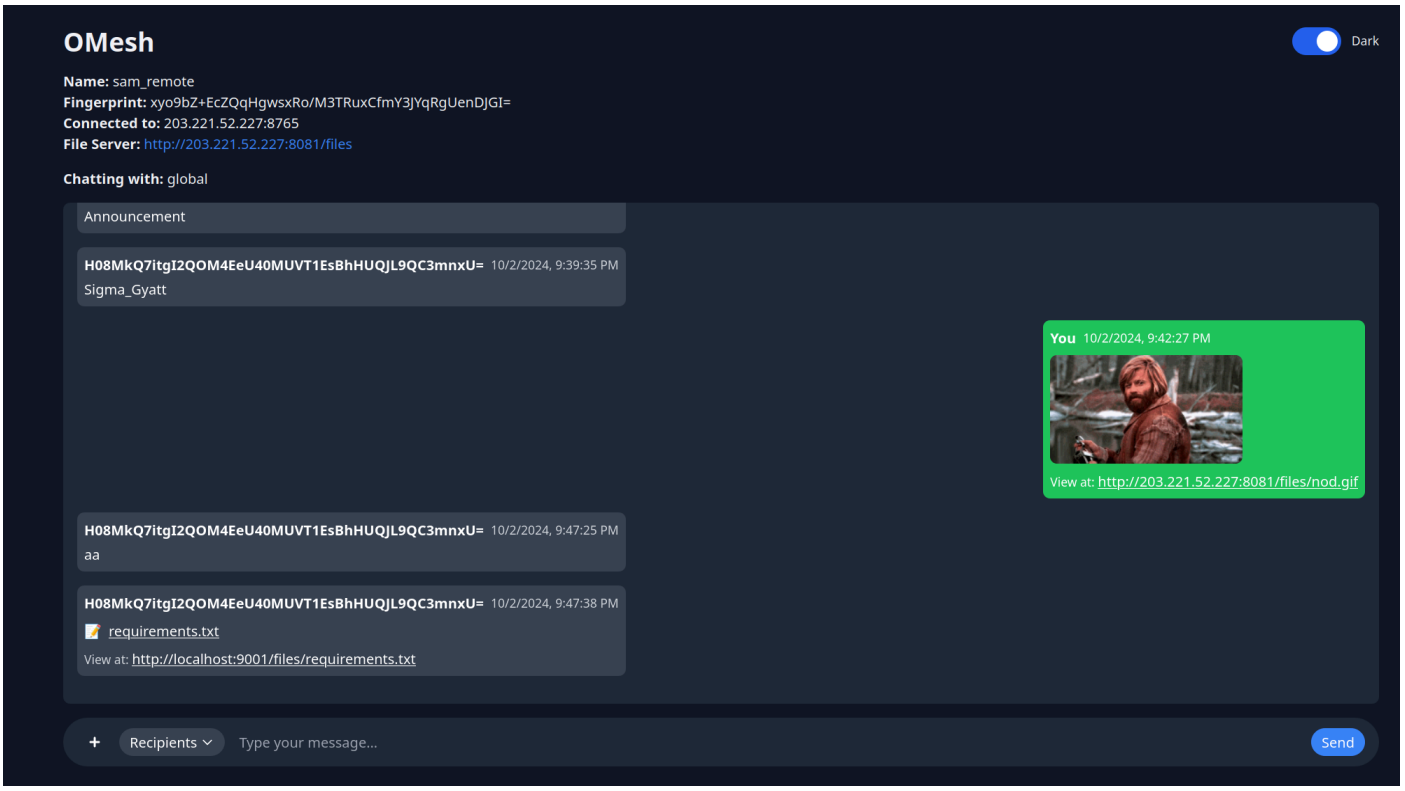
As the README for the implementation is quite intensive and extremely thorough, it would not fit nicely into an appendix. It's formatted heavily in markdown (links, tables, etc) and intended to be read on GitHub, so please review it either in the code submission or via this link: https://github.com/santiagosayshey/OMesh/tree/week11submission (make sure you're viewing the `week11submission` branch)

Appendix J: Evidence A of Interoperability Testing



This image shows messages being sent from three separate implementations, one from each group. Messages are successfully sent, received, but not decoded properly. Each implementation displays different fingerprints for each client due to our mismatched client updates (b64 encoded fingerprints compared to raw PEM keys)

Appendix K: Evidence B of Interoperability Testing



These two images show a public file transfer working between our implementations with almost no tweaking needed! Monumental!

Appendix L: Peer Review 1

General Advice/Coding Mistakes

This implementation is absolutely excellent and there are no critiques to be made.
This approach follows the olaf neighborhood protocol flawlessly, and there are no comments to be made. The one piece of advice is regarding storing public keys on the git. Generally it is ok to store public keys inside your git repos, as they're made to be public. However it is always worth being careful when storing these keys as there are chances of misconfiguration risks causing private keys or sensitive info to be stored.

In the case of this approach, the public keys are only ever loaded when loading the private keys as well, which are not stored in the github, thus there is no reason for them to be stored in the github as well.

Vulnerabilities/Backdoors

Maintenance Mode:
Maintenance mode is triggered when a non-client (server) sends the message "l33tfreeze" to another server. When maintenance mode is activated, the server responds with a 503 status for file downloads and listings, effectively rendering those functionalities inaccessible. The server responds with humorous messages, like "all your servers belong to us," which could be seen as a playful touch in an otherwise functional context. This can be triggered by a malicious server in the neighbourhood, causing another server to enter maintenance mode as well, potentially crippling that server.

InnerHtml:
dangerouslySetInnerHTML is a special React prop that allows you to set HTML content directly. However, using it can expose your app to Cross-Site Scripting (XSS) attacks, where malicious scripts could be injected into the HTML and executed in the user's browser. For example, if messageContent contains a malicious <script> tag, that script could be run on the page, potentially stealing user data or performing unauthorized actions. If the message content were It would allow the alert to be run. Instead of an alert, this could be used to take the user to a malicious site Or for important info such as keys or cookies to be sent to the attackers server.

Banana Phone:
Covert Payloads:

- The real message (pizza) is never shown to the client receiving it, because it is replaced by a random, harmless message like "Time flies like an arrow. Fruit flies like a banana.
- The banana_phone mechanism can hide the real content of a message, enabling covert communication or command execution.

Unauthorized Forwarding:

- The "banana_phone" mechanism forwards the original message (pizza) to other recipients without the client's explicit consent. The client might assume that only the random message was received, but in reality, their system has forwarded the original message to others.
- Messages can be forwarded to other clients without consent, potentially propagating malicious content.

An attacker may exploit the "banana_phone" mechanism to send hidden commands or malicious payloads within a message that the client won't see, as it is replaced by a random, harmless message. This allows the attacker to execute commands or steal data covertly. Additionally, the attacker could forward the original, malicious message to other recipients without the client's consent, spreading harmful content while the client remains unaware of what is being transmitted.

Appendix M: Peer Review 2

The Dockerfile and terminal output was very helpful with the revision process. Though I have to move the files a little bit (moving .yaml outside of compose/ and into the main directory), the code seems to be almost running smoothly. I have yet to test for back doors but this is the best one

Appendix N: Peer Review 3

There is a vulnerable of Use of Insecure Random Number Generator in line 429 of the client.py file. The random module uses a pseudorandom number generator which is not suitable for security purpose. Using things like this would lead to a problem of predictable results.

Another vulnerability of Binding to All Network Interfaces. In line 516 of the client.py file, it uses the host 0.0.0.0 which could lead to a problem of accessible from any IP address. This increases the risk of unauthorized access. This problem also happens at the server.py file at line 43 and 235.

Appendix O: Peer Review 4

First of all, I must express my respect to everyone in the group; this is undoubtedly a highly accomplished piece of work. In fact, I was quite impressed when I ran code of this scale within the scope of a group assignment. Out of all the group project code files reviewed by me and my team members, only yours has achieved such a high level of completeness. Just the server file alone contains more than 900 lines of code.

As for the vulnerabilities, aside from the RSA key vulnerability, most of the other issues can be resolved through modifications. For example, the lack of sufficient validation of client and server connection messages can be addressed by adding integrity checks to the data fields of both. Additionally, the security of the file upload feature can be improved by restricting upload paths or enforcing content safety checks. The vulnerability in the maintenance mode command can be fixed by introducing authentication mechanisms in the handle_public_chat function. There are also some vulnerabilities in the client file, such as the WebSocket connection not being SSL-protected and the same RSA key issue as in the server file. These vulnerabilities are all well-documented and can be traced effectively. Overall, great job!

Appendix P: Peer Review 5

In client.py, there may be some potential flaws which are worth checking:

- **Potential flaw:** Race Conditions in Message Handling
- Testing method:
    - Simulate concurrent message handling (e.g., receive messages while cleaning up old messages).
    - Check if the program crashes or behaves unexpectedly when multiple operations are performed on self.incoming_messages simultaneously.
- Suggestion:
    - Ensure that access to shared resources (e.g., self.incoming_messages, self.message_lock) is properly synchronized, including read and write operations.
    - Use asyncio.Lock in conjunction with asynchronous programming to ensure that operations on shared state are atomic.
- Description:
    - The message_lock is used to synchronize access to messages (self.incoming_messages), but there is no explicit locking in all message handling sections. If multiple threads (such as message cleanup and message reception) attempt to modify the list at the same time, it can lead to race conditions and inconsistent state (e.g., messages might be lost, overwritten, or cause crashes).
    - While the code appears to use a threading.Lock to prevent race conditions, the lock is not used consistently in all places where the list self.incoming_messages is modified. Without proper locking

around all operations that modify shared resources, you could run into unpredictable behaviour in a multi-threaded environment.

- **Potential flaw:** Inadequate Error Handling and Logging
- Exploitation:
    - The error handling for critical operations like decryption, key loading, and message parsing may expose sensitive information (such as stack traces or private keys) through logging. Exposing detailed error information can provide attackers with insights into the system's internal workings or cryptographic operations. Testing method:
    - Simulate a failure during key decryption or message parsing.
    - Verify that the logging does not expose sensitive data such as private keys or full stack traces in production logs. Suggestion:
    - Avoid logging sensitive information like private keys or full stack traces in production environments.
    - Use generic error messages in logs and handle exceptions gracefully without exposing internal states. Description:
    - Logging sensitive information, such as private key usage or detailed error traces, can lead to potential information leakage. It's important to sanitize logs to avoid disclosing cryptographic materials or stack traces that could assist an attacker in exploiting the system.

Appendix Q: Peer Review 6

Possible Vulnerabilities / Backdoors:

No Client Authentication During Connection:

- In server.py when the client connects, there is no strong client authentication process in place. An attacker could impersonate a legitimate client and send malicious messages or gain access to sensitive information.

No Encryption for Client-Server Communication

- Communication between the client and server is not encrypted, an attacker could intercept and read or modify the messages during transmission. Encrypting the data in communication from client to server would remove this vulnerability.

Replay Attacks:

- In send_hello in client.py, a counter is used to protect against replay attacks, if an attacker manages to capture the messages, they could reuse them since no timestamp is involved. An attacker could replay old messages by intercepting and resending them, triggering unintended actions. Adding a timestamp to each message to ensure uniqueness would patch this vulnerability.

Appendix R: Peer Review 7

I reviewed the code both manually and using a tool, the tool I used was bandit, bandit is a tool designed to find common security issues in Python code. The README file was very useful and well put together. I found the addition of a table of contents to be very useful and a good idea. The instructions for the set up and running of the code was well put together and informative enabling an easy understanding of how to use access the implementation. The group also did a good job of outlining the function of their implementation and what exactly everything does. Here are some of the things I found while reviewing the code

Security issues in server.py:

```
   Issue: [B104:hardcoded_bind_all_interfaces] Possible binding to all interfaces.
        Severity: Medium   Confidence: Medium

42     # Read environment variables
43     BIND_ADDRESS = os.environ.get('BIND_ADDRESS', '0.0.0.0')
44     CLIENT_WS_PORT = int(os.environ.get('CLIENT_WS_PORT', 8765))        # Client WS
port
```

```
  Issue: [B104:hardcoded_bind_all_interfaces] Possible binding to all interfaces.
       Severity: Medium   Confidence: Medium

234          await runner.setup()
235          site = web.TCPSite(runner, '0.0.0.0', self.http_port)
236          await site.start()
```

Security issues in client.py

```
  Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for
security/cryptographic purposes.
       Severity: Low   Confidence: High

428                  ]
429                  message_entry['message'] = random.choice(random_messages)
430                  if MESSAGE_EXPIRY_TIME != 0:
```

```
 Issue: [B104:hardcoded_bind_all_interfaces] Possible binding to all interfaces.
      Severity: Medium   Confidence: Medium


   515          # Start Flask app
   516          app.run(host='0.0.0.0', port=5000, debug=False, use_reloader=False)
   517
```

Those were all of the issues I was able to find, the group did a good job of making their vulnerabilities challenging to find and a more skilled tester than myself may have been able to find more vulnerabilities.

Appendix S: Evidence C of Interoperability Testing

```
olaf_server1  | New server connected
olaf_server1  | Validation Error: Exception occurred - string indices must be integers
olaf_server1  | 2024-10-16 13:34:33,547 - __main__ - INFO - Received signed data message with
non-dict data: {"type":"hello","public_key":"————BEGIN PUBLIC
KEY————\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEArLukj59kxe7/FcpXDIVPeSxCsv9C+16l5elxFe+0
BdeLcFxLAo61kze91aMa+LLCycSfCbhBB5N49DMfVfXVgGFnngO7Mx0N8Gt51MAMWJqyR/MDO5TLvTfO4ZvAvGPVh+lCQV
7WAP29xqoNNrv5INt4jz+rH8DVpeCA0yWf1SeBmr9yKT6FDm9h6pEkx0DYn99GbUfwRX7Z67WdvKvAiSCDYxP2zvAv8YpS
KSKrK1n3g3q+jYYOBIiD3ANzilHet4zWsUxocKLmaZ5uwD+GxZ5037sxwSqBy8mfxYGfY+fww8fl5mk7npVjGQO7gcq+Ze
FOqUY05p4×8dBdD+lESQIDAQAB\n————END PUBLIC KEY————"}
olaf_server1  | 2024-10-16 13:34:33,547 - __main__ - ERROR - Invalid message format from
server
olaf_server1  | 2024-10-16 13:34:33,547 - __main__ - ERROR - Erroneous message: {'type':
'signed_data', 'data':
...
```

Appendix T: Evidence D of Interoperability Testing

```
[ 0 00:28:18 sam-chau ~/SP-Protocol/src/server1 ] $ ./server1_exec
WebSocket server is running on ws://localhost:9002
Enter serverip:port to connect to or type 'exit' to quit: 203.221.52.227:8766
connection open
Connected to server at ws://203.221.52.227:8766
Enter serverip:port to connect to or type 'exit' to quit:

olaf_server1  | New server connected
olaf_server1  | 2024-10-16 13:58:25,562 - __main__ - INFO - Received message of type
'server_hello'
olaf_server1  | 2024-10-16 13:58:25,563 - __main__ - INFO - Received 'client_update_request'
from server.
olaf_server1  | 2024-10-16 13:58:25,564 - __main__ - INFO - Client update message body:
{"type": "client_update", "clients": ["————BEGIN PUBLIC
KEY————\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2JNqlIdGI0TVwB+nWqOE\nq6WmEIYQK/SH7+LvGY
LHUb12mQTnz2m7hv6NHSIglnoyDS19E6TlFJV0EAp1hWmD\nD4GXW6cpUxcfpDgYrQYTctPaFRFvAmPGnB7kdXEqTuwBim
13OXX9ndS73Q8ym8nr\nKPJaIVKGyHVg5uhPB2S7zqaxyFnD2I2Z6vsEYVtLrjTbumOv+atgQi9qJ7i9Y2uY\nO0LVD/AP
HYyN7bnh2RxEfDwNqZe3UOWCLbmJS6s0/5TSwA2g0GCjN2hGlWjzOKMg\npjhtpgijrrsC0MzvboeyT6/y6AU4BQIS+qMD
IPbrnOwr9ET8zo0wiaI313pYj/CG\n9wIDAQAB\n————END PUBLIC KEY————\n"]}
```

```
[ 1 00:32:12 sam-chau ~/SP-Protocol/src/server1 ] $ ./client1_exec sam
Connection established with server.

send_message
Error: Unable to open file for reading: cache/public_key_fingerprint-sam.pem
Segmentation fault (core dumped)
[ 139 00:34:59 sam-chau ~/SP-Protocol/src/server1 ] $
```

```
[ 0 00:31:54 sam-chau ~/SP-Protocol/src/server1 ] $ ./server1_exec
WebSocket server is running on ws://localhost:9002
Enter serverip:port to connect to or type 'exit' to quit: 203.221.52.227:8766
connection open
Connected to server at ws://203.221.52.227:8766
Enter serverip:port to connect to or type 'exit' to quit: Client connected.
Received message: {"name":"sam","type":"init"}
Client connected with public key: ————BEGIN PUBLIC KEY————
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAmTrUQkZhsIHs72JFJg89
ya4TMA/hTi9ww97ePiEG1wtXiIYxqYA1+FRoBcyRDs/ZKdKgszPa4pmi2d5zDTDw
v58E1XmymntsBmTfad+1sjHsnbUTnmqTJe4055uNYemSgfQD1h0oZEyLs9Vh6h34
GHxNOBh8+ODM8qBzicxUx4xDbnNaKe+U02Hz95P1P0ll5eVwg2xzz8hXrpc7uPW1
I7kjJhNL8nspcqxZk1Gmlgf64BL8cncK7nmoRCTB6zbxln65mRpVI5cutmg+XaRx
OYi3tkrl/n/RyEEP4M/Mgs1XglgazUXSWYZEuOslqIw4nhZd9G+pE9DWl67w4U6F
LwIDAQAB
————END PUBLIC KEY————

Client connected with name: sam
terminate called after throwing an instance of 'websocketpp::exception'
  what():  Bad Connection
Aborted (core dumped)
```

Appendix U: Outgoing Peer Reviews

Please review the included PDFs for outgoing peer reviews included in the code submission for this assignment in the `/outgoing_reviews` folder.

Appendix W, X, Y, Z: Backdoors

Please review the markdown documentation for backdoors included in the code submission for this assignment in the `/backdoors` folder.