Group33

1. Server-Side Code (server.py)

Several essential features for managing client-server communication via WebSockets are included in the server-side code. The WebSocket handler, which controls incoming client connections, handles disconnections, and processes messages, is one crucial component. Keeping up real-time communication requires this.

Key Function: websocket_handler

```python
async def websocket_handler(self, websocket):
    CLIENT_IP, CLIENT_PORT = websocket.remote_address
    prev_counter = 0
    client_public_key = b""

    logger.info(f"[+] {CLIENT_IP}:{CLIENT_PORT} connected")

    try:
        async for message in websocket:
            if not message:  # Check if the message is empty
                logger.warning(f"[-] Empty message received from {CLIENT_IP}:{CLIENT_PORT}")
                await websocket.close(code=4004, reason="Empty message received")
                break

            json_message = None

            # If the recieved message is not able to be decoded then close the connection
            try:
                json_message = json.loads(message)

            except json.JSONDecodeError as e:
                logger.error(f"[#] Could not decode message from {CLIENT_IP}:{CLIENT_PORT}: {e}")
                await websocket.close(code=4000, reason="Invalid JSON format")
                break
```

Source reference:SecureProgrammingGroup42

Explanation:

This function manages the server's WebSocket communication. This is how it operates:

1. The server uses a WebSocket connection to listen for messages from clients.
2. It processes incoming messages by verifying that they are in a proper JSON format.

3. The server will terminate the connection with an error code if the message is incorrect or empty.

Potential Challenges:

Message Validation: To avoid problems like incorrect requests resulting in unexpected crashes or vulnerabilities, it is essential to make sure that only legitimate, correctly designed messages are executed.

Security Risk: Unauthorised users may be able to connect to the WebSocket and engage with the system if this handler is not properly authenticated.

Comparison to our code:

Our server tracks clients using a dictionary that maps client fingerprints to their WebSocket connections. This is a simple yet effective way to manage connected clients and their public keys.

```
async def send_client_list(websocket):
    client_list = {
        "type": "client_list",
        "servers": [
            {
                "address": "localhost",
                "clients": list(client_public_keys.keys())
            }
        ]
    }
    await websocket.send(json.dumps(client_list))
```

Source reference:SecureProgrammingGroup33

Client Management: Client fingerprints are mapped to WebSocket connections in the prior code as well, but it has more sophisticated client list management features, such as ways to communicate and update the list to other servers. Adding stronger client management will enhance our implementation.

Improvements:

- **Client Management Enhancements**: As seen in the preceding code, provide a mechanism to update the client list dynamically. This might include adding methods to broadcast the list of connected clients or transmitting it upon request. Scaling the system to many servers may be aided by this.Enhancements to Client Management: As seen in the preceding code, provide a mechanism to update the client list dynamically. This might include adding methods to broadcast the list of connected clients or transmitting it upon request. Scaling the system to many servers may be aided by this.
- **RSA Key Storage**:We may wish to include a mechanism to persist the public keys, such writing them to a file or database, even if we keep them in memory. When managing server restarts or reconnections, this might be beneficial.

Advantages of Our Client Management:

- **Simplicity**: For a limited number of clients, the system remains lightweight and manageable via the straightforward use of a dictionary to handle client connections.

Disadvantages of Our Client Management:

- **No Persistence**: Our system would lose track of connected clients after a restart or crash if client public keys were not persistent, necessitating that all clients rejoin and re-register.

3. Thoughts on Embedded Backdoors

The backdoor is a secret way to get around standard encryption or authentication procedures, usually concealed in trustworthy code. These may be purposefully included in a number of methods, including bypasses in crucial verification parts, secret keys, and hardcoded passwords.

Potential Backdoor Example:

It is possible to introduce a basic backdoor into the client's authentication scheme. For instance, if a certain circumstance is satisfied, a hardcoded check in client_app_vulnerable.py might let any user to get around authentication rather than correctly checking the credentials:

```
def login(connection):
    cursor = connection.cursor()

    while True:
        fingerprint = input(f"{BOLD}Enter your fingerprint: {END}")
        password = input(f"{BOLD}Enter your password: {END}")

        query = f"""SELECT private_key FROM users WHERE fingerprint = '{fingerprint}' AND password = '{password}'"""

        try:
            cursor.execute(query)
            db_private_key = cursor.fetchone()

            if not db_private_key:
                print(f"{RED}{BOLD}Incorrect fingerprint or password{END}")
                continue

            private_key = serialization.load_pem_private_key(
                db_private_key[0].encode('utf-8'),
                password=None
            )

            rsa = RSA(private_key)

            return rsa
```

Source reference:SecureProgrammingGroup42

The fingerprint "backdoor_fingerprint" in this code would provide access without confirming the password, granting unauthorised access to the system. It is quite difficult to identify this backdoor without a comprehensive analysis due to the code's simplicity.