HW06: Secure Rock-Paper-Scissors Protocol

Simone Nolé
1940213

December 4, 2025

Abstract

This report presents the design and implementation of a secure protocol for playing "Rock, Paper, Scissors" over a network. The system utilizes a Trusted Third Party (Server) architecture to ensure fairness and prevent cheating. Security is guaranteed through a hybrid cryptosystem that uses RSA for authentication and key-exchange, and AES-GCM for secure session communication.

Contents

1 Introduction

The goal of this assignment is to implement a networked version of the game *Roshambo* (Rock, Paper, Scissors) that is secure against network attackers and dishonest players. The implementation involves two clients (alice.py and bob.py) and a central server (boss.py). For key-exchange, it was required to generate for each element (Bob, Alice, and Boss) a private key and a certificate. This has been done by launching the following commands:

```
# keys's generation for Boss (Server)
openssl req -x509 -newkey rsa:4096 -keyout server_key.pem -out server_cert.pem -days 365 -nodes

# keys's generation for Alice
openssl req -x509 -newkey rsa:4096 -keyout client1_key.pem -out client1_cert.pem -days 365 -nodes

# keys's generation for Bob
openssl req -x509 -newkey rsa:4096 -keyout bob_key.pem -out bob_cert.pem -days 365 -nodes
```

Listing 1: OpenSSl to generate certificates and private keys

In addition, each player and the server are containerized thanks to Docker.

2 Threat Model and Security Goals

In designing the protocol, we considered the following threats:

- **Network Sniffing (Eve):** An external attacker trying to read the moves or the outcome.
- **Man-in-the-Middle:** An attacker trying to intercept or modify messages.
- **Dishonest Players (Cheating):** Alice changing her move after seeing Bob's, or vice versa.
- **Replay Attacks:** An attacker re-sending old valid messages to simulate a move.

To mitigate these, the protocol guarantees:

1. **Confidentiality:** Achieved via AES-256 encryption.
2. **Integrity, Non-Repudiation & Authenticity:** Achieved via RSA Digital Signatures and AES-GCM Tags.
3. **Freshness:** Achieved via Timestamps and Nonces to prevent replay attacks.

3 Protocol Design

The architecture follows a Client-Server model where the server acts as the judge.

3.1 Phase 1: Authentication and Key Exchange

Before the game starts, a secure channel is established.

1. Client generates a random 256-bit AES Session Key.
2. Client encrypts the key with Server's RSA Public Key.
3. Client signs the timestamp and the encrypted key with their RSA Private Key.
4. Server verifies the signature and decrypts the session key.

3.2 Phase 2: The Game (Encrypted Session)

Once authenticated, all traffic is encrypted using AES-GCM.

1. Alice sends her move (encrypted). Server stores it but does not reveal it.
2. Bob sends his move (encrypted).
3. Once the Server has both moves, it calculates the winner.
4. Server sends the result to both players.

4 Implementation Details

The solution is implemented in Python using the `socket` and `cryptography` libraries. Also, each Bob and Alice function, to choose randomly the value (Rock, Paper, Scissors) it has been used the `Secrets` library for a random number in [0,2].

5 Anti-Cheating Mechanism

The requirement "secure operations against possible cheating" is satisfied by the **Commitment Scheme** inherent in the Trusted Third Party architecture:

- **Secrecy:** When Alice sends "Rock", it is encrypted. Bob cannot sniff the packet to decide his move accordingly.
- **Immutability:** Once the Server receives Alice's move, Alice cannot change it. The Server waits for Bob's move before computing the result.

6 Infrastructure and Virtualization

To meet the requirement of implementing the protocol on distinct virtual machines and simulating a realistic network environment, it has been adopted a **Container-based architecture** using Docker.

This approach ensures process isolation and provides a separate network stack for each actor (Alice, Bob, and the Server), effectively simulating three distinct machines connected via a local network.

6.1 Docker Build Environment

The application environment for all three services (`boss`, `alice`, and `bob`) is defined by a single `Dockerfile`.

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . /app
```

Listing 2: Dockerfile

- **Base Image:** We utilize `python:3.10-slim` for a minimal, secure, and lightweight operating system, reducing the overall container size.
- **Dependency Management:** The `pip install` command ensures that all necessary Python libraries, including `cryptography` and `pycryptodomex` (or equivalents), are installed prior to execution.
- **Context:** The application code is copied into the `/app` working directory, allowing the `docker-compose` service commands (e.g., `python boss.py`) to run immediately upon container startup.

6.2 Docker Compose Configuration

The orchestration of the environment is managed via `docker-compose`, which defines three services:

- **Boss (Server):** Acts as the central Trusted Third Party. It is accessible to other containers via the hostname `boss`.
- **Alice & Bob (Clients):** Two distinct containers that execute the client logic. They wait for the server to be ready before starting (via `depends_on`).

Below is the configuration used to deploy the environment:

```
services:
  # Service 1: The Trusted Third Party (Server)
  boss:
    build: .
    command: python boss.py
    ports:
      - "8080:8080"
    hostname: boss
    restart: always

  # Service 2: Client Alice
  alice:
    build: .
    command: python alice.py
    depends_on:
      - boss
    stdin_open: true # Keeps container alive for interaction
    tty: true

```

```
  # Service 3: Client Bob
  bob:
    build: .
    command: python bob.py
    depends_on:
      - boss
    stdin_open: true
    tty: true
```

Listing 3: Docker Compose Configuration

6.3 Network Simulation

Docker automatically creates a virtual bridge network (default subnet). Within this network, **Service Discovery** is handled via internal DNS. Instead of hardcoding IP addresses (e.g., `192.168.1.x`), clients connect to the server using the hostname `boss`. This abstraction mimics a real-world DNS resolution scenario and makes the code environment-agnostic.

- **Server Address:** `boss` (Resolved automatically by Docker DNS)
- **Port:** 8080 (Mapped for internal communication)

7 Execution and Logs

Below is an example of the execution trace.

7.1 Alice Output

```
LOADED PUBLIC KEY <cryptography.hazmat.bindings._rust.openssl.rsa.RSAPublicKey object at
    0x7ff6be65bbf0> of FILENAME server_cert.pem
Connected to the server
sent who i am
Waiting for message...
Waiting for message...
Server acknowledged game message
Waiting for message...
I played: Paper, Bob played: Rock. Winner: Alice
Do you want to play again? (yes/no)
yes
Waiting for message...
Server acknowledged game message
Waiting for message...
I played: Paper, Bob played: Scissors. Winner: Bob
Do you want to play again? (yes/no)
no
Exiting the game.
```

Listing 4: Alice output

7.2 Bob Output

```
>>python bob.py
LOADED PUBLIC KEY <public_key> of FILENAME server_cert.pem
Connected to the server
sent who i am
Waiting for message...
```

```
Waiting for message...
Server acknowledged game message
Waiting for message...
I played: Rock, Alice played: Paper. Winner: Alice
Do you want to play again? (yes/no)
yes
Waiting for message...
Server acknowledged game message
Waiting for message...
I played: Scissors, Alice played: Paper. Winner: Bob
Do you want to play again? (yes/no)
no
Exiting the game.
```

Listing 5: Bob output

7.3 Server (Boss) Output

```
>>python boss.py
LOADED PUBLIC KEY <cryptography.hazmat.bindings._rust.openssl.rsa.RSAPublicKey object at
    0x7fb49cbe2af0> of FILENAME alice_cert.pem
LOADED PUBLIC KEY <cryptography.hazmat.bindings._rust.openssl.rsa.RSAPublicKey object at
    0x7fb49cbe2a50> of FILENAME bob_cert.pem
Server listening on ('0.0.0.0', 8080)
[NEW CONNECTION] ('127.0.0.1', 34874) connected.
[('127.0.0.1', 34874)] Waiting for message...
[('127.0.0.1', 34874)] Cleartext message received: auth
Valid Signature: Is Alice!
[('127.0.0.1', 34874)] Authenticated! Session Key stored.
[('127.0.0.1', 34874)] Waiting for message...
[('127.0.0.1', 34874)] Game message received with value: Paper
Waiting for the other player to make a move...
Waiting for the other player to make a move...
[NEW CONNECTION] ('127.0.0.1', 48780) connected.
[('127.0.0.1', 48780)] Waiting for message...
[('127.0.0.1', 48780)] Cleartext message received: auth
Valid Signature: is Bob!
[('127.0.0.1', 48780)] Authenticated! Session Key stored.
[('127.0.0.1', 48780)] Waiting for message...
[('127.0.0.1', 48780)] Game message received with value: Rock
Both players have made their moves: Alice(Paper) vs Bob(Rock)
Determining winner: Alice(paper) vs Bob(rock)
Game result determined: Alice
Sending game result to Bob...
[('127.0.0.1', 48780)] Waiting for message...
Both players have made their moves: Alice(Paper) vs Bob(Rock)
Determining winner: Alice(paper) vs Bob(rock)
Game result determined: Alice
Sending game result to Alice...
[('127.0.0.1', 34874)] Waiting for message...
[('127.0.0.1', 34874)] Game message received with value: Paper
Waiting for the other player to make a move...
Waiting for the other player to make a move...
[('127.0.0.1', 48780)] Game message received with value: Scissors
Both players have made their moves: Alice(Paper) vs Bob(Scissors)
Determining winner: Alice(paper) vs Bob(scissors)
Game result determined: Bob
Sending game result to Bob...
[('127.0.0.1', 48780)] Waiting for message...
Both players have made their moves: Alice(Paper) vs Bob(Scissors)
Determining winner: Alice(paper) vs Bob(scissors)
Game result determined: Bob
Sending game result to Alice...
[('127.0.0.1', 34874)] Waiting for message...
[DISCONNECT] ('127.0.0.1', 34874) disconnected.
[DISCONNECT] ('127.0.0.1', 48780) disconnected.
```

Listing 6: Boss (Server) output

8 Conclusion

The implemented protocol successfully allows a secure game of Rock Paper Scissors. The use of RSA for the handshake ensures that only authorized clients can play, while AES-GCM ensures that moves remain confidential until the showdown, preventing any form of cheating.

9 Code

9.1 rps.py

```
import secrets

rps = {0:"Scissors", 1:"Rock", 2:"Paper"}

def rock_paper_shissors_secure():
    return rps[secrets.randbelow(3)]

if __name__ == "__main__":
    count_rps = {"Shissors":0, "Rock":0, "Paper":0}
    print(("-"*10)+"Testing"+("-"*10))
    for i in range(1000000000):
        match rock_paper_shissors_secure():
            case "Shissors":
                count_rps["Shissors"] +=1
            case "Rock":
                count_rps["Rock"] +=1
            case "Paper":
                count_rps["Paper"] +=1
    print(f"Shissors = {count_rps['Shissors']}, Rock = {count_rps['Rock']}, Paper = {count_rps['Paper']}")
    print(("-"*10)+"Finish"+("-"*10))
```

Listing 7: rps.py code that choose safelly and randomly Rock, Paper or Scissors

9.2 tcp$_j$*son.py*

```
import json
import struct

def recvall(sock, n):
    """Helper function to receive EXACTLY n bytes or die trying"""
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None # the connection was closed
        data += packet
    return data

def receive_json(sock):

    header = recvall(sock, 4)
    if not header:
        return None # the connection was closed

    msg_length = struct.unpack('>I', header)[0]

    payload_bytes = recvall(sock, msg_length)

    try:
        packet_dict = json.loads(payload_bytes.decode('utf-8'))
        return packet_dict
    except json.JSONDecodeError:
```

```python
        print("ERROR: Malformed JSON received!")
        return None

def send_json(sock, packet_dict):

    json_str = json.dumps(packet_dict)

    data_bytes = json_str.encode('utf-8')

    msg_length = len(data_bytes)

    header = struct.pack('>I', msg_length)

    sock.sendall(header + data_bytes)
```

Listing 8: tcp.py code that wrappers sendall and recvall function for TCP sockets and manages JSON communications properly

9.3 enc.py

```python
import os
import time
import base64
import json
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
from cryptography import x509
from cryptography.hazmat.backends import default_backend
from tcp_json import send_json
from tcp_json import receive_json


def encrypt_message(msg_plaintext, key):
    nonce = os.urandom(12)

    algorithm = algorithms.AES(key)
    mode = modes.GCM(nonce)
    cipher = Cipher(algorithm, mode, backend=default_backend())

    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(msg_plaintext.encode('utf-8')) + encryptor.finalize()

    tag = encryptor.tag

    return nonce, ciphertext, tag

def decrypt_message(nonce, ciphertext, tag, key):

    try:
        algorithm = algorithms.AES(key)
        mode = modes.GCM(nonce, tag)
        cipher = Cipher(algorithm, mode, backend=default_backend())

        decryptor = cipher.decryptor()
        decrypted_data = decryptor.update(ciphertext) + decryptor.finalize()

        return decrypted_data.decode('utf-8')

    except Exception as e:
        return "ERROR: Someone touched the file!"

def load_private_key(filename):
    with open(filename, "rb") as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
            backend=default_backend()
        )
```

```python
    return private_key

def load_public_key_from_cert(filename):
    # 1. read bytes from file.pem
    with open(filename, "rb") as cert_file:
        cert_data = cert_file.read()

    # 2. laod the certificate as X.509
    cert = x509.load_pem_x509_certificate(cert_data, default_backend())

    # 3. extract the public key from the certificate
    public_key = cert.public_key()

    print(f"LOADED PUBLIC KEY {public_key} of FILENAME {filename}") #Debug

    return public_key

def is_timestamp_valid(received_timestamp, ttl=60):
    server_time = time.time()

    delta = server_time - received_timestamp

    if delta < -2.0:
        print(f"[SECURITY ALERT] Timestamp coming from the future! Difference: {delta:.2f}s. Clocks not synchronized or attack.")
        return False

    if delta > ttl:
        print(f"[SECURITY ALERT] Packet expired! Old by {delta:.2f}s (Max: {ttl}s). Possible Replay Attack.")
        return False

    return True

def send_json_encrypted(message, conn, client_id, session_key):
    nonce, ciphertext, tag = encrypt_message(json.dumps(message), session_key)
    message_encrypted = {
        "Symm-encrypted":"y",
        "client_id":client_id,
        "nonce": base64.b64encode(nonce).decode('utf-8'),
        "ciphertext": base64.b64encode(ciphertext).decode('utf-8'),
        "tag": base64.b64encode(tag).decode('utf-8')
    }
    send_json(conn, message_encrypted)

def receive_and_decrypt_json_encrypted(conn, session_key, message=None):
    if message is None:
        encrypted_json = receive_json(conn)
    else:
        encrypted_json = message

    # receive_json can return None if the peer disconnected or JSON was malformed
    if encrypted_json is None:
        return None

    try:
        # Convert from Base64 String -> Original Bytes
        nonce = base64.b64decode(encrypted_json['nonce'])
        ciphertext = base64.b64decode(encrypted_json['ciphertext'])
        tag = base64.b64decode(encrypted_json['tag'])

        # 2. Decryption
        decrypted_str = decrypt_message(nonce, ciphertext, tag, session_key)

        # If decrypt_message returns an error string, handle it robustly
        if not isinstance(decrypted_str, str) or decrypted_str.startswith("ERRORE") or decrypted_str.startswith("ERROR"):
            print("Decryption failed (Tag mismatch or wrong key)")
            return None

        return json.loads(decrypted_str)

    except (KeyError, ValueError, json.JSONDecodeError) as e:
```

```
        print(f"Error in encrypted protocol: {e}")
        return None
```

Listing 9: enc.py contains many aux functions for symmetric encryption, asymmetric encryption and sending/receiving functions that wrappers send$_{j}son/receive_{j}sonfunctions$

9.4 boss.py (Server

```
import socket
import threading
import base64
from time import sleep
from enc import load_private_key
from enc import load_public_key_from_cert
from enc import is_timestamp_valid
from tcp_json import receive_json
import struct
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from enc import send_json_encrypted
from enc import receive_and_decrypt_json_encrypted

HOST = "0.0.0.0"
PORT = 8080

alice_session_key = 0
bob_session_key = 0

alice_value = {"value": "", "count": 0}
bob_value = {"value": "", "count": 0}

keys_db = {
    "alice": load_public_key_from_cert("alice_cert.pem"),
    "bob": load_public_key_from_cert("bob_cert.pem")
}

def determine_winner(alice_move, bob_move):
    a = alice_move.lower()
    b = bob_move.lower()
    print(f"Determining winner: Alice({a}) vs Bob({b})")  # Debug

    valid_moves = ["rock", "paper", "scissors"]

    if a not in valid_moves or b not in valid_moves:
        return "Error: Invalid Move"

    if a == b:
        return "Draw"

    if (a == "rock" and b == "scissors") or \
       (a == "scissors" and b == "paper") or \
       (a == "paper" and b == "rock"):
        return "Alice"

    return "Bob"

def handle_auth(msg, conn, addr):
    global alice_session_key, bob_session_key
    timestamp = msg["timestamp"]
    if (is_timestamp_valid(timestamp) == False):
        print("WARNING! timestamp not valid. possible replay attack")
        return False

    encrypted_blob = base64.b64decode(msg["encrypted_key"])
    signature_bytes = base64.b64decode(msg["signature"])

    server_private_key = load_private_key("server_key.pem")
    decrypted_session_key = server_private_key.decrypt(
```

```
        encrypted_blob,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    #print(f"decrypted session key {decrypted_session_key}")

    try:
        # Ricostruisce i dati originali (Key decifrata + Timestamp ricevuto)
        data_to_verify = decrypted_session_key + struct.pack('>d', timestamp)

        if (msg["client_id"] == "alice"):
            keys_db["alice"].verify(
                signature_bytes,
                data_to_verify,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
            print("Valid Signature: Is Alice!")
            alice_session_key=decrypted_session_key

        elif (msg["client_id"] == "bob"):
            keys_db["bob"].verify(
                signature_bytes,
                data_to_verify,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
            print("Valid Signature: is Bob!")
            bob_session_key=decrypted_session_key

        # Usa la session key per comunicare con alice/bob
        message = {
            "type" : "game"
        }
        send_json_encrypted(message, conn, "server", decrypted_session_key)
        return decrypted_session_key

    except Exception:
        print("Invalid Signature: Someone is liying!")
        message = {
            "type" : "you're a liar!"
        }
        send_json_encrypted(message, conn, "server",decrypted_session_key)
        return None

def handle_game(msg, conn, addr, session_key):

    if session_key is None:
        print(f"[{addr}] No session key available for game message from {msg.get('client_id')}")
        return

    print(f"[{addr}] Game message received with value: {msg.get('value')}")
    global alice_value, bob_value
    if msg.get("client_id") == "alice":
        alice_value["value"] = msg.get("value")
        alice_value["count"] += 1
    elif msg.get("client_id") == "bob":
        bob_value["value"] = msg.get("value")
        bob_value["count"] += 1
    else:
        print(f"[{addr}] Unknown client_id: {msg.get('client_id')}")
        return

```

```python
    response = {
        "type": "game ack",
        "value": msg.get("value")
    }
    send_json_encrypted(response, conn, "server", session_key)
    while True:
        if (alice_value["count"]==bob_value["count"]):
            print(f"Both players have made their moves: Alice({alice_value['value']}) vs Bob({bob_value['value']})")
            winner = determine_winner(alice_value["value"], bob_value["value"])
            print(f"Game result determined: {winner}")
            result_message = {
                "type": "game result",
                "winner": winner,
                "alice_value": alice_value,
                "bob_value": bob_value
            }

            if (msg.get("client_id") == "alice"):
                print(f"Sending game result to Alice...")
            elif (msg.get("client_id") == "bob"):
                print(f"Sending game result to Bob...")
            send_json_encrypted(result_message, conn, "server", session_key)

            break
        print("Waiting for the other player to make a move...")
        sleep(1)


def handle(conn, addr):
    print(f"[NEW CONNECTION] {addr} connected.")

    current_session_key = None

    while True:
        try:

            print(f"[{addr}] Waiting for message...")
            wrapper_msg = receive_json(conn)

            if not wrapper_msg:
                # Client disconnected
                break

            # --- first branch (Auth) ---
            if wrapper_msg.get('Symm-encrypted') == "n":

                print(f"[{addr}] Cleartext message received: {wrapper_msg.get('type')}")

            if wrapper_msg.get("type") == "auth":

                current_session_key = handle_auth(wrapper_msg, conn, addr)

                if current_session_key is None:
                    print(f"[{addr}] Authentication failed.")
                    break
                else:
                    print(f"[{addr}] Authenticated! Session Key stored.")

            # --- second branch: Encrypted messages (Game) ---
            else:
                if current_session_key is None:
                    print(f"[{addr}] ERROR: Attempt to send encrypted message without auth.")
                    break

                decrypted_msg = receive_and_decrypt_json_encrypted(conn, current_session_key, wrapper_msg)

                if decrypted_msg is None:
                    print(f"[{addr}] Error decrypting or disconnected.")
                    break

```

```
                match decrypted_msg.get('type'):
                    case "game":
                        handle_game(decrypted_msg, conn, addr, current_session_key)
                    case "disconnect":
                        print(f"[{addr}] Disconnect request.")
                        break
                    case _:
                        print(f"[{addr}] Unknown message type: {decrypted_msg.get('type')}")

        except ConnectionResetError:
            print(f"[{addr}] Connection Reset.")
            break
        except Exception as e:
            print(f"[{addr}] Generic error in loop: {e}")
            break

    conn.close()
    print(f"[DISCONNECT] {addr} disconnected.")

def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((HOST, PORT))
        s.listen()
        print("Server listening on", (HOST, PORT))
        while True:
            conn, addr = s.accept()
            t = threading.Thread(target=handle, args=(conn, addr), daemon=True)
            t.start()

if __name__ == "__main__":
    main()
```

Listing 10: boss.py is basically the server that implements everything that is needed to communicate to each clients in a secure way and elaborates the winner

9.5 alice.py

```
import secrets
import socket
import time
import struct
import base64
from time import sleep
from enc import load_private_key
from enc import load_public_key_from_cert
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from tcp_json import send_json
from cryptography import x509
from cryptography.hazmat.backends import default_backend
from rps import rock_paper_shissors_secure
from enc import send_json_encrypted
from enc import receive_and_decrypt_json_encrypted


HOST = 'boss'
PORT = 8080
session_key = secrets.token_bytes(32)

def load_private_key(filename):
    with open(filename, "rb") as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
            backend=default_backend()
        )
```

```
    return private_key

def load_public_key_from_cert(filename):
    # 1. read bytes from file.pem
    with open(filename, "rb") as cert_file:
        cert_data = cert_file.read()

    # 2. laod the certificate as X.509
    cert = x509.load_pem_x509_certificate(cert_data, default_backend())

    # 3. extract the public key from the certificate
    public_key = cert.public_key()

    print(f"LOADED PUBLIC KEY {public_key} of FILENAME {filename}") #Debug

    return public_key


server_public_key= load_public_key_from_cert("server_cert.pem")

def sendWhoIAm(socket):
    #encrypting the session key with the server public key
    encrypted_blob = server_public_key.encrypt(
        session_key,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    #signing the key with the timestamp with the alice private key
    alice_private_key = load_private_key("alice_key.pem")
    timestamp = time.time()
    timestamp_bytes = struct.pack('>d', timestamp)
    data_to_sign = session_key+timestamp_bytes
    signature = alice_private_key.sign(
        data_to_sign,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    handshake_packet = {
        "Symm-encrypted":"n",
        "client_id": "alice",
        "type": "auth",
        "timestamp": timestamp,
        "encrypted_key": base64.b64encode(encrypted_blob).decode('utf-8'),
        "signature": base64.b64encode(signature).decode('utf-8')
    }

    send_json(socket, handshake_packet)

#TODO
def game_result(message, conn):
    # placeholder: process a game result message from server
    # print(f"Game result received: {message}")
    my_value = message.get("alice_value")
    bob_value = message.get("bob_value")
    winner = message.get("winner")
    print(f"I played: {my_value['value']}, Bob played: {bob_value['value']}. Winner: {
    winner}")
    sleep(1)
    print("Do you want to play again? (yes/no)")
    answer = input().strip().lower()
    if answer != "yes":
        print("Exiting the game.")
        return False
    else:game(message, conn)
    return True
```

```python

#TODO
def game(message, conn):
    value = rock_paper_shissors_secure()
    message = {
        "client_id" : "alice",
        "type" : "game",
        "value" : value
    }
    #print("i'm inside the game function!")
    send_json_encrypted(message, conn, "alice",session_key)
    return True

def handle(message, conn):
    msg_type = message.get("type")
    match msg_type:
        case "you are a liar":
            print("damn he found me! I have to escape")
            return False
        case "game":
            return game(message, conn)
        case "game result":
            return game_result(message, conn)
        case "game ack":
            print("Server acknowledged game message")
            return True

def main():

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as conn:
        conn.connect((HOST, PORT))
        print("Connected to the server")

        sendWhoIAm(conn)
        print("sent who i am")

        # receive initial message and then keep receiving inside the loop
        while True:
            print("Waiting for message...")
            # Do not pass 0 here; pass no message (or None) so the function reads from
    socket
            message = receive_and_decrypt_json_encrypted(conn, session_key)
            if not message:
                print("No message received, closing connection")
                break
            # print(f"message received : {message}")
            if handle(message, conn) == False:
                break



if __name__ == "__main__":
    main()
```

Listing 11: alice.py enstablishes the connection to the server, proves that she is the real Alice and play the game. It's basically the exact same code that Bob has

9.6 bob.py

```python
import secrets
import socket
import time
import struct
import base64
from time import sleep
from enc import load_private_key
from enc import load_public_key_from_cert
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
```

```
from cryptography.hazmat.primitives.asymmetric import padding
from tcp_json import send_json
from cryptography import x509
from cryptography.hazmat.backends import default_backend
from rps import rock_paper_shissors_secure
from enc import send_json_encrypted
from enc import receive_and_decrypt_json_encrypted


HOST = 'boss'
PORT = 8080
session_key = secrets.token_bytes(32)

def load_private_key(filename):
    with open(filename, "rb") as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None,
            backend=default_backend()
        )
    return private_key

def load_public_key_from_cert(filename):
    # 1. read bytes from file.pem
    with open(filename, "rb") as cert_file:
        cert_data = cert_file.read()

    # 2. laod the certificate as X.509
    cert = x509.load_pem_x509_certificate(cert_data, default_backend())

    # 3. extract the public key from the certificate
    public_key = cert.public_key()

    print(f"LOADED PUBLIC KEY {public_key} of FILENAME {filename}") #Debug

    return public_key


server_public_key= load_public_key_from_cert("server_cert.pem")

def sendWhoIAm(socket):
    #encrypting the session key with the server public key
    encrypted_blob = server_public_key.encrypt(
        session_key,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    #signing the key with the timestamp with the alice private key
    # use Bob's private key for signing
    bob_private_key = load_private_key("bob_key.pem")
    timestamp = time.time()
    timestamp_bytes = struct.pack('>d', timestamp)
    data_to_sign = session_key+timestamp_bytes
    signature = bob_private_key.sign(
        data_to_sign,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    handshake_packet = {
        "Symm-encrypted":"n",
        "client_id": "bob",
        "type": "auth",
        "timestamp": timestamp,
        "encrypted_key": base64.b64encode(encrypted_blob).decode('utf-8'),
        "signature": base64.b64encode(signature).decode('utf-8')
```

```python
    }

    send_json(socket, handshake_packet)

#TODO
def game_result(message, conn):
    # placeholder: process a game result message from server
    # print(f"Game result received: {message}")
    my_value = message.get("bob_value")
    alice_value = message.get("alice_value")
    winner = message.get("winner")
    print(f"I played: {my_value['value']}, Alice played: {alice_value['value']}. Winner:
    {winner}")
    sleep(1)
    print("Do you want to play again? (yes/no)")
    answer = input().strip().lower()
    if answer != "yes":
        print("Exiting the game.")
        return False
    else:game(message, conn)
    return True

#TODO
def game(message, conn):
    value = rock_paper_shissors_secure()
    message = {
        "client_id" : "bob",
        "type" : "game",
        "value" : value
    }
    #print("i'm inside the game function!")
    send_json_encrypted(message, conn, "bob",session_key)
    return True

def handle(message, conn):
    msg_type = message.get("type")
    match msg_type:
        case "you are a liar":
            print("damn he found me! I have to escape")
            return False
        case "game":
            return game(message, conn)
        case "game result":
            return game_result(message, conn)
        case "game ack":
            print("Server acknowledged game message")
            return True

def main():

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as conn:
        conn.connect((HOST, PORT))
        print("Connected to the server")

        sendWhoIAm(conn)
        print("sent who i am")

        # receive initial message and then keep receiving inside the loop
        while True:
            print("Waiting for message...")
            # Do not pass 0 here; pass no message (or None) so the function reads from
    socket
            message = receive_and_decrypt_json_encrypted(conn, session_key)
            if not message:
                print("No message received, closing connection")
                break
            #print(f"message received : {message}")
            if handle(message, conn) == False:
                break



if __name__ == "__main__":
```

```
    main()
```

Listing 12: bob.py enstablishes the connection to the server, proves that she is the real Bob and play the game. Basically it is the exact same code that Alice has.