Rock Paper Scissors - Secure Network Protocol

https://github.com/andreaduca/cs_hw6

This exercise is based on implementing a Cryptographic Commitment Scheme: if Alice sends "ROCK" first, Bob (if malicious) can see it, calculate that "PAPER" wins, and send “PAPER”. So, Bob cannot see Alice move untill he commits his move.

Commitment Schemes:

1. Commit: Alice takes her move, and hash it using also her secret key to hide her move to Bob and be able on the end to reveal it. She sends to Bob her hashed move.
Bob cannot open it, but he knows Alice has played something.

2. Play: Bob plays his move effectively "blind".

3. Reveal: Alice gives Bob the key. Bob verifies the Hash, sees the move and checks who won.

NOTE: I must use a Nonce. If Alice just hashes the word "ROCK", Bob can pre-calculate the hashes for "ROCK", "PAPER", and "SCISSORS" and cheat. By adding a random string (Nonce) to the move, the hash becomes unpredictable.

- - -

A Client-Server Architecture Implementation

I will designate Bob as the Server (Listener) and Alice as the Client.

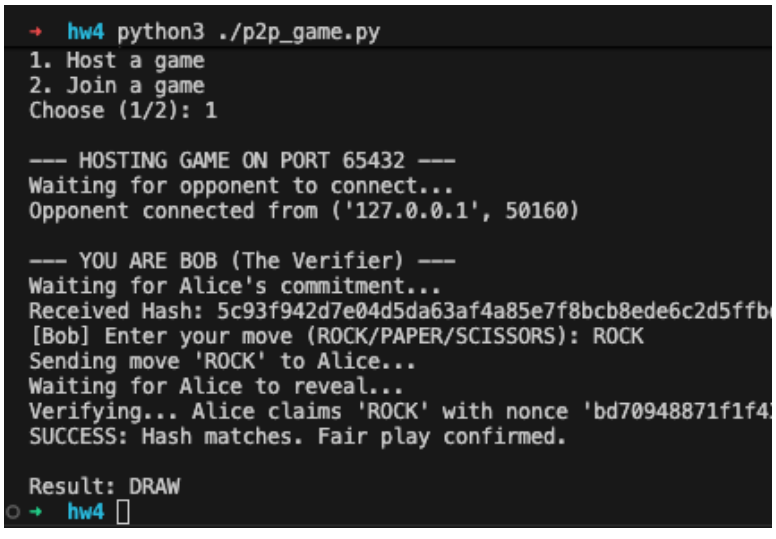
1. Alice picks a move (e.g., "ROCK") and generates a random secret (Nonce).

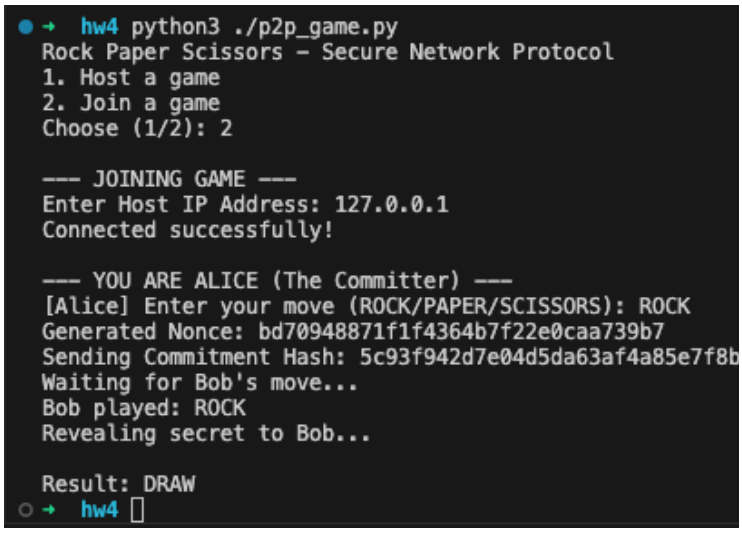
2. Alice calculates Hash(Move + Nonce) and sends this Hash to Bob.

3. Bob receives the Hash. He cannot know the move, so he picks his move (e.g., "PAPER") and sends it to Alice.

4. Alice receives Bob's move. She now sends her original Move and Nonce to Bob to prove she didn't cheat.

5. Bob calculates Hash(Received_Move + Received_Nonce).
- If it matches the Hash from Step 2, the move is valid.
- Bob calculates the winner.





Details

Socket Architecture: `SOCK_STREAM` (TCP)

I initialized the soket with:
`socket.socket(socket.AF_INET, socket.SOCK_STREAM)`

Address Family (AF_INET): This tells the OS I'm using IPv4 (standard IP addresses), allowing the game to work over a local network or the internet.

Socket Type (SOCK_STREAM): This selects the TCP Protocol.

Why is this important? TCP is a *reliable* protocol. In this game, if Alice sends the hash of her move, every single byte must arrive in the correct order.
If a UDP socket were used, packets might be lost or arrive out of order, which would make the cryptographic hash verification fail.

Connection Handling: The Handshake

The logic is asymmetric:
Host is a passive listener. `bind(('0.0.0.0', PORT))` to claim the port number PORT on all networks interfaces (Wi-Fi, Ethernet, localhost) on the computer. It keep listening untill a guess connects.
Guest is the active initiator. `connect((target_ip, PORT))` performs the TCP 3-Way Handshake. It reaches out to the IP provided by the user. If an Host is listening at that IP, the connection is established.

The Secure Exchange Mechanism (Application Layer)

Once the connection is established, the stateful protocol for the game starts, this means the order of operations is fixed and prearrange.

1.**Serialization (Text to Bytes)** Sockets cannot send Python strings; they can only send raw bytes.

2.**Alice's Commitment** (Alice's Move): sha256(move + nonce)

3.**Reveal: SHA256(ReceivedMove+ReceivedNonce).** If it matches with Alice's commitment, this mathematically guarantees that Alice did not change her move after seeing Bob's play.

Conclusions

The application operates on the Transport Layer using TCP sockets (SOCK_STREAM) to ensure data integrity, which is critical for cryptographic verification.

The connection logic follows a standard Client-Server model where the Host binds to a port and enters a blocking `accept()` state, while the Guest initiates the 3-way handshake via `connect()`.

The application layer protocol uses blocking synchronous calls (`sendall` and `recv`), enforcing a strict turn-based state machine.

Data serialization is handled via UTF-8 encoding. The security relies on Alice revealing her secret in a specific format (Move:Nonce), allowing Bob to locally re-compute the SHA256 hash and verify it against the commitment received earlier in the session.