Muhammad Sohaib Appicoders Blockchain Technical Test

Q#3

Q3. Fiat-to-Token Conversion on-chain

To securely implement a Fiat-to-Token conversion process on-chain, several important aspects need to be considered, such as price oracles, handling slippage/rounding, and ensuring auditability of the process.

1. Price Oracle Usage

Price oracles are essential for providing off-chain data to the blockchain. In the case of Fiat-to-Token conversion, you will need an oracle to feed the current Fiat-to-Token exchange rate (such as USD to your native token or any other fiat currency to ERC20 tokens).

Popular Oracle Solutions: Chainlink, Band Protocol, and API3 are widely used for fetching off-chain data securely. These platforms aggregate data from trusted sources (e.g., centralized exchanges, forex APIs) and ensure the data is consistent and accurate.

Security Considerations:

Ensure the oracle is decentralized to prevent manipulation by a single point of failure.

Use multiple oracle sources (aggregation) to avoid reliance on a single provider, reducing the risk of manipulation or downtime.

Verify oracle integrity by introducing thresholds or consensus for the data to be considered valid (e.g., using median values from multiple oracles).

2. Handling of Rounding/Slippage

When converting Fiat to Tokens, rounding and slippage can cause discrepancies between the actual amount a user receives and what they expect.

Rounding: Due to the difference in precision between fiat and token decimal places, rounding is inevitable. This should be handled by specifying a fixed number of decimals for the conversion. For instance, in ERC20 tokens, you could standardize to 18 decimal places, which may not directly align with fiat's precision.

Solution: Implement a rounding mechanism that rounds down or up based on the specific business rules you prefer. A common approach is to round down (floor function) to avoid overshooting the value for users.

Slippage: Slippage occurs when the price changes between the time the transaction is initiated and when it is finalized.

Solution: Implement a slippage tolerance in the smart contract, such as a 1% tolerance on the price oracle's rate. If the price changes by more than this threshold, the transaction will fail, and the user will need to re-trigger the transaction.

Example: Use the formula expected\_amount \* (1 - slippage\_tolerance) to calculate the maximum amount the user is willing to pay for a given token amount.

3. Ensuring Auditability

Auditability ensures that all Fiat-to-Token transactions can be traced, verified, and validated by any third party.

Events: Emit events for all key actions in the smart contract, such as:

When a Fiat-to-Token conversion occurs.

When the oracle price is updated.

When a transaction fails due to slippage.

Emit detailed logs with the transaction data, including the fiat amount, token amount, exchange rate, slippage, and transaction timestamp.

Transaction Logs: Store all conversion transactions in a publicly accessible contract storage or decentralized storage (e.g., IPFS, Arweave) for transparency. This allows external auditors and users to review and verify each conversion event.

Auditable Smart Contract: Use well-tested and open-source code, and ensure that the contract is open for inspection. You could also have it verified on Etherscan and regularly perform smart contract audits.

Q#4

Q4. Competition-Based Payout System using Smart Contracts

A Competition-Based Payout System involves distributing rewards to participants based on their rankings or achievements in the competition. Here's how you could design it:

1. Token Pooling Mechanism

In this system, the rewards are distributed from a pool of tokens. These tokens are collected either through entry fees or sponsors, and then allocated to winners.

Entry Fees: Participants pay a fee in tokens to enter the competition. The collected tokens are stored in the contract and distributed based on the competition results.

Smart Contract Design: Upon entry, each participant’s payment is recorded, and the contract ensures that the total pool balance increases. The contract can track the amount of tokens a participant contributed to the pool.

Sponsor Contributions: If the competition is sponsored, sponsor tokens can be added to the pool. A function can be created to accept sponsor donations.

2. Distribution Logic

Once the competition ends, the distribution logic comes into play to reward the winners.

Ranking System: Use a ranking system to determine the winner(s). The ranking could be based on performance metrics like time, score, or points. The smart contract should have a function that assigns rankings to participants at the end of the competition.

Example: The contract could have a list of participants and their respective scores, and the rankings can be determined by sorting them from highest to lowest score.

Reward Distribution: Once rankings are determined, distribute rewards to the winners from the token pool.

Reward Allocation: The contract could distribute rewards based on the predefined prize pool distribution:

Example: 1st place gets 50%, 2nd place gets 30%, and 3rd place gets 20% of the total pool.

Implement this via a mapping that keeps track of the prize share for each participant.

Equal Distribution: If it's a random drawing competition, then tokens can be distributed equally among the winners or based on a probability system coded into the smart contract.

Smart Contract Design:

Use a function to calculate and distribute payouts automatically once the competition concludes.

Emitting events after each payout allows transparency in the system.

3. Optional Dispute Mechanism or Admin Verification

In case of disputes (e.g., when a participant believes the competition was unfair), an optional verification or dispute mechanism can be implemented.

Admin Verification: Allow an admin to manually verify the results and adjust the payouts. This could be done using the Ownable modifier or specific admin functions in the contract.

Example:

solidity

function adminAdjustReward(address winner, uint256 amount) external onlyOwner {

require(amount > 0, "Amount must be positive");

// Transfer reward to winner

rewardToken.transfer(winner, amount);

emit AdminAdjustment(winner, amount);

}

Dispute Resolution: A dispute system could allow participants to raise issues related to the competition's fairness. For example, an arbitrator could be chosen (maybe from a list of trusted third parties), and the contract could allow the arbitrator to review and make decisions regarding the dispute.

Example:

solidity

function raiseDispute(uint256 competitionId) external {

// Logic for raising a dispute

}

function resolveDispute(uint256 competitionId, bool decision) external onlyArbitrator {

// Resolve dispute based on decision

}

Smart Contract Design: The contract could include a mapping to store disputes and their status (open, resolved, etc.).

By using these designs, you can effectively create a secure and auditable Fiat-to-Token conversion system and a Competition-Based Payout System with smart contracts. The use of events, oracle data, and transparent logging ensures that everything remains verifiable, while the payout logic and dispute mechanism add flexibility and fairness.