Milestone 2 - Simple Smart-Contract

Solidity

V2.1 Scope

Implement ERC20 Interface from scratch. (Take **IERC20** interface)
NOT ERC20 Already ready implementation.
Implement Transfer, BalanceOf, and other IERC20 functions yourself.
https://docs.openzeppelin.com/contracts/5.x/extending-contracts

V2.2 Scope

Beyond normal ***IERC20*** interface functions, here is the list of additional non-functional requirements:

1. **Token Holders** that own **more than 0.1%** of the supply can start voting for a price change of the token.
2. Each token holder that owns **more than 0.05% (minTokenAmount)** of the supply can vote for a particular price via function: "***function vote(uint256 price)***".
3. Time to vote is defined by a constant in the contract ***"timeToVote"*** or this value could be configured in the constructor
4. The voting power is the amount of tokens a voter holds. For instance, if users A and B hold 100 tokens each, and person C holds 250 tokens, then even if both A and B vote for the same price, the price that C voted for would be selected.
5. The opportunity to buy/sell the token for ether, based on the current token price. New tokens are minted on the contract, in exchange for ether in the "buy" function. In the "sell" function, tokens are burned, and ether is returned from the contract's pool (balance).
6. Configurable percentage by admin of the fee for the "Buy" and "Sell" functions. Fee is collected and burned weekly.
7. For the purposes of **V2.2** Scope, forbid to transfer/buy/sell tokens if you have voted and a voting is still pending.

Functional Requirements:

1. Use typechain to generate types of your smart contracts in the tests
2. The contract should be upgradeable
3. EndVoting function can be called by anyone but the rules enforce it to only be called after "timeToVote" time has elapsed since the beginning of the voting.
4. Contracts should be deployed to the testnet (Sepolia), and verified on sepolia.etherscan.

Tests

Implement the following tests for the contract above

1. StartVoting should change “votingStartedTime”, “votingNumber”, emit “VotingStarted event.
2. Vote transactions should pass or fail depending on the condition that the user should at least have **0.05%** of token supply on the balance.
3. Token transfers don’t allow users to double-spend tokens on voting
   We should have a protection against the following scenario:
   a. User A votes for a price with X tokens on balance
   b. User B hesitates to vote, User B balance = Y
   c. User A transfers X amount of tokens to User B, User B new balance = X + Y
   d. User B votes for a price with X + Y tokens on balance
4. The tests should **travel in time** using **hardhat**.

1. **All the tests should check that a corresponding event is emitted!**
2. **Tests should be configured to print values in either of the formats: eth, gwei, wei. (Applicable to eth and tokens).**

Tech Stack and Setup

1. Solidity for Smart Contracts
2. Ethers.js for Tests on TypeScript, or Nethereum for tests on F#.
   a. If .NET language is F#, we recommend using RedDuck’s Type Provider.
3. Hardhat for running blockchain

Guide (how to develop with hardhat)

Build and Deploy Smart Contracts using Hardhat

FAQ

Q: Как сделать так, чтобы автоматически, по наступлению определенной даты/времени, контракт сделал действие X?
A: Никак. Контракты (программы) автоматически не могут работать. Все функции, которые вызываются в программах, рано или поздно прекращают работать, ибо транзакции, в которых функции выполняются, имеют ограничение по времени выполнения / трудоемкости. Это называется "gas limit" - почитайте в теории Ethereum в этом разделе. Написать цикл, который сам раз в какой-то промежуток что то вызывает, невозможно. Это возможно не в солидити, а в оффчейн скрипте. Короче говоря, функции надо вызывать вручную, допустим через ethers.js

Однако, эту задачу могут решать chainlink keepers или Gelato AutoTask, но им надо немного заплатить.

Topics that have been learnt:

1. **Formatting (wei, gwei, eth, etc.)**
2. **IERC20, ERC20**
3. **Solidity**
4. **Ethers.js + hardhat**
5. **Gas**
6. **Memoization pattern**

Problem solved:

- **Problem with the cycles, O(N) functions that are dependent on users.**