Smart Contracts Exercise 03: ERC-20 CTU Token

1 Introduction

Tokens in the Ethereum ecosystem are smart contracts that implement a standardized interface. They are designed to represent various assets digitally. These assets can range from financial instruments like company shares and stablecoins (e.g., USDC, DAI) to governance tokens that allow holders to vote on decisions in decentralized projects (e.g., Uniswap's UNI). Tokens can also enable artists to tokenize their works and sell them as unique digital items (NFTs), represent collectibles in games, or are used for digital identity or access to services. Depending on the use case, there are different types of tokens, each serving distinct purposes. Below are three common types of tokens:

1. **Fungible Tokens**

   Fungible tokens (*ERC-20 Tokens*) are interchangeable and have exactly the same value. Each unit of a fungible token is identical to another unit. Examples are cryptocurrencies, utility tokens or governance tokens.

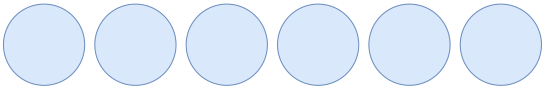


Figure 1: Fungible Tokens

2. **Non-Fungible Tokens (NFTs)**

   Non-fungible tokens (*ERC-721 Tokens*) are unique and cannot be exchanged on a one-to-one basis. They are used to represent ownership of unique items such as digital art, collectibles, and real estate. Each token is uniquely identifiable by an ID.

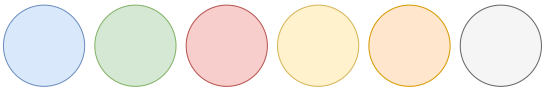


Figure 2: Non-Fungible Tokens

3. **Multi-Tokens**

   Multi-tokens (*ERC-1155 Tokens*) combine the properties of both fungible and non-fungible tokens. They allow for the creation of multiple token types within a single contract, providing flexibility for various use cases.

Figure 3: Multi-Tokens

In this exercise, you'll create your own ERC-20 token contract following the specified standard, and then attempt to hack it. For more information about tokens, check out this animated video: What Are NFTs and How Can They Be Used in Decentralized Finance? You'll also work with your own NFTs in future exercises.

Project Setup

You have two options for working with this exercise. Using Docker container or local installation. Choose the one that best fits your preferences.

1.1 Using Docker with VS Code

This option uses Docker to create a development environment with all the necessary tools and dependencies pre-installed.

Prerequisites:

- **Docker** - A platform for developing, shipping, and running applications in containers.
- **Visual Studio Code** - A lightweight but powerful source code editor.
- **Dev Containers** - An extension to VS Code that lets you use a Docker container as a full-featured development environment.

Setting Up the Project:

1. Visit the following GitLab repository and clone it to your local machine.
2. Open the repository folder in VS Code.
3. When prompted, click "Reopen in Container" or use the command palette (F1) and run `Dev Containers: Reopen in Container`.

1.2 Local Setup

If you prefer working directly on your machine without Docker, you can set up the development environment locally.

Prerequisites

- **Node.js**: `https://nodejs.org/en/` - An open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- **NPM**: Node Package Manager, which comes with Node.js.

Open your terminal and run the following commands to verify the installations:

```
$ node -v
$ npm -v
```

Both commands should return the installed version numbers of Node.js and NPM respectively. Node.js provides the runtime environment required to execute JavaScript-based tools like Hardhat, while NPM is used to manage the packages and dependencies needed for development.

Setting Up the Project

1. Visit the following GitLab repository and clone it to your local machine.
2. Open a terminal and navigate to the project directory.
3. Install the project dependencies by running `npm install`.

2 Specification: ERC-20 Token

The ERC-20 standard was first proposed by Fabian Vogelsteller and Vitalik Buterin in November 2015. The token specification defines the interface that a smart contract must implement to be ERC-20 compliant. It is important to note that it **does not specify the actual implementation**. It is the most widely used standard with more than 1.5 million smart contracts on the mainnet implementing it.

Example functionalities ERC-20 provides:

- Transfer tokens from one account to another.
- Get the current token balance of an account.
- Get the total supply of the token available on the network.
- Approve whether an amount of token from an account can be spent by a third-party account.

If a Smart Contract implements the following methods and events, it can be called an ERC-20 Token Contract. Once deployed, it will be responsible for keeping track of the created tokens on Ethereum. To see the full specification, visit EIP20 proposal.

Methods

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value)
  public returns (bool success)
function approve(address _spender, uint256 _value) public returns (bool success)
function allowance(address _owner, address _spender)
  public view returns (uint256 remaining)
```

Events

```
  event Transfer(address indexed _from, address indexed _to, uint256 _value)
  event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

OpenZeppelin

OpenZeppelin provides an open-source library for secure smart contract development. It is built on a solid foundation of community-vetted code. It is good practice to use standardized implementations like those from OpenZeppelin. Documentation about available contracts made by OpenZeppelin can be found here. The actual implementations of the contracts are available on GitHub. OpenZeppelin contracts can be installed using npm and imported directly into a contract. The ERC20 implementation by OpenZeppelin is a standard recognized by the official EIP20 documentation. You can find the implementation here. It is a common practice to use the ERC20 implementation by OpenZeppelin when creating ERC20 token contracts, instead of explicitly implementing the ERC20 interface inside the contract. However, for the educational purpose of this exercise, you will implement the ERC20 contract by yourself. The implementation of the CTU Token contract using OpenZeppelin can be seen in the code below.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// Import OpenZeppelin's ERC20 implementation
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

/**
 * @title CTUToken
 * @dev A custom implementation of an ERC-20 Token using OpenZeppelin's library.
 */
contract CTUToken is ERC20 {
    // Define the initial supply: 1,000,000 tokens with 18 decimal places
    uint256 private constant INITIAL_SUPPLY = 1_000_000 * 10 ** 18;

    /**
     * @dev Constructor that initializes the ERC-20 token with a name and symbol,
     * and mints the total supply to the deployer's address.
     */
    constructor() ERC20("CTU Token", "CTU") {
        // Mint the initial supply to the deployer of the contract
        _mint(msg.sender, INITIAL_SUPPLY);
    }
}
```

3 CTU Token

To complete the CTU Token contract and pass all the associated tests, you need to implement the required functionality in the `contracts/CTUToken.sol` file. Look for the sections marked with #TODO comments and ensure that all the specified requirements below are met.

Token Details

- **Name**: Set the token name to `"CTU Token"`.
- **Symbol**: Set the token symbol to `"CTU"`.
- **Decimals**: Use `18` decimal places to align with the standard Ether denomination.
- **Total Supply**: Initialize the total supply to `1,000,000` tokens.

Events

- **Transfer Event**: Emit a `Transfer` event whenever tokens are transferred, including zero-value transfers.
- **Approval Event**: Implement and emit an `Approval` event when an allowance is set via the `approve` function.

Errors

- **TransferToZeroAddress**: Attempting to transfer to the zero address.
- **InsufficientBalance(uint256 requested, uint256 available)**: Account does not have enough balance.
- **ApproveToZeroAddress**: Attempting to approve the zero address as a spender.
- **TransferExceedsAllowance(uint256 requested, uint256 allowance)**: Trying to transfer an amount exceeding the current allowance.

State Variables

- **Balances Mapping**: Maintain a `mapping(address => uint256)` to track the token balance of each account.
- **Allowances Mapping**: Use a nested `mapping(address => mapping(address => uint256))` to manage allowances, enabling accounts to authorize others to spend tokens on their behalf.

Constructor

- Assign the entire `totalSupply` to the contract deployer's address upon deployment.

Core Functions

- **name()**: Return the name of the token.
- **symbol()**: Return the token symbol.
- **decimals()**: Return the number of decimal places (`18`).
- **totalSupply()**: Return the total supply of tokens in existence.

- **balanceOf(address account)**: Return the token balance of the specified account.
- **transfer(address to, uint256 value)**:
  - Ensure the recipient address is not the zero address.
  - Verify that the sender has a sufficient balance.
  - Update the sender's and recipient's balances accordingly.
  - Emit a `Transfer` event.
- **approve(address spender, uint256 value)**:
  - Ensure the spender address is not the zero address.
  - Set the allowance for the spender.
  - Emit an `Approval` event.
- **allowance(address owner, address spender)**: Return the remaining number of tokens that the spender is allowed to spend on behalf of the owner.
- **transferFrom(address from, address to, uint256 value)**:
  - Ensure neither the sender nor the recipient is the zero address.
  - Verify that the source account has a sufficient balance.
  - Check that the caller has enough allowance.
  - Update the balances of the source and recipient accounts.
  - Decrease the allowance accordingly.
  - Emit a `Transfer` event.

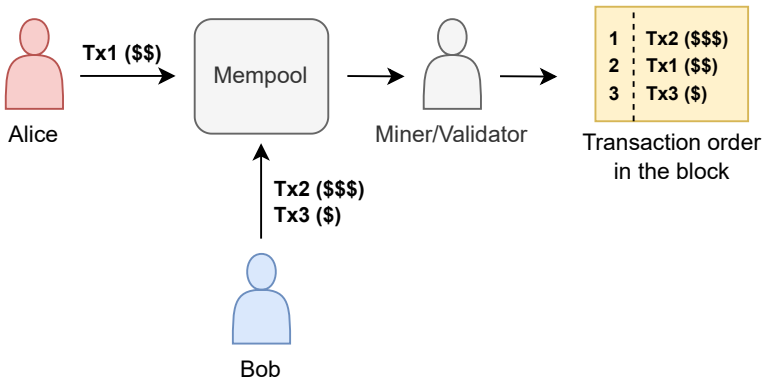
To fulfill the exercise, you need to pass all the provided tests. You can run the tests with the following command:

```
$ npx hardhat test
```

4 Frontrunning Attack

Did your contract pass all the tests? Congratulations! It is now fully tested and ready for deployment to the mainnet. But is it really?

When you submit a transaction, it first enters the public mempool where anyone can see it. To get your transaction included in the blockchain, you need to pay a fee to validators. Validators typically choose transactions that offer the highest profit rather than processing them in chronological order. This means transactions submitted earlier might be included later, while those submitted more recently could be processed sooner. Note: this is a simplified explanation - we'll cover this topic more thoroughly in a future exercise about MEV.



The following table shows the attack vector for our CTU Token smart contract, where the attacker Bob manages to steal CTU tokens from Alice due to a frontrunning attack.

Step	Actor	Action	Description
1	Alice	Approve Bob	Alice approves Bob to spend 100 CTU tokens on her behalf using the `approve` function.
2	Bob	Monitor Mempool	Bob monitors the mempool for pending transactions, specifically looking for Alice's approval transaction.
3	Alice	Change Approval	Alice decides to change Bob's allowance from 100 CTU to 200 CTU and initiates a new `approve` transaction.
4	Bob	Frontrun Transaction	Upon detecting Alice's new approval transaction in the mempool, Bob quickly submits a `transferFrom` transaction to utilize the existing 100 CTU allowance before Alice's transaction is mined. Bob sets a higher gas price to prioritize his transaction.
5	Network	Transaction Ordering	Due to the higher gas price, Bob's `transferFrom` transaction is mined before Alice's new `approve` transaction. This allows Bob to transfer 100 CTU tokens before the allowance is updated.
6	Alice	Approval Mined	Alice's `approve` transaction is mined, increasing Bob's allowance to 200 CTU.
7	Bob	Exploit Increased Allowance	Bob now has an increased allowance of 200 CTU and can transfer an additional 200 CTU from Alice's account, totaling 300 CTU gained.

Review the file `scripts/attack.js` where this attack is implemented and understand it. Then run the attack using the command

```
$ npx hardhat run scripts/attack.js --config priority.mempool.js
```

For this attack, we use a special configuration file for Hardhat, where we set that a new block is confirmed every 500ms and transactions are selected for inclusion in the blockchain based on priority fees. In our simulation, each block contains only one transaction for simplicity. However, this does not protect against frontrunning attacks, as transactions with higher fees will simply be included in earlier blocks.

Task:

Modify the CTU Token smart contract to prevent vulnerability to frontrunning attacks.

- **Tip:** To display all blocks in the simulation, uncomment the last line of the main function.

- **Tip:** Use these functions for easier debugging: `displayState`, `printMempool`, `printLastBlockInfo`, `printAllBlocksInfo`.
- **Tip:** You might need to change the approval process completely.