MIT World Peace University

Attack Research and Documentation
Fourth Year B. Tech, Semester 8

Solidity Programming of simple Smart Contracts

Lab Assignment 2

Prepared By

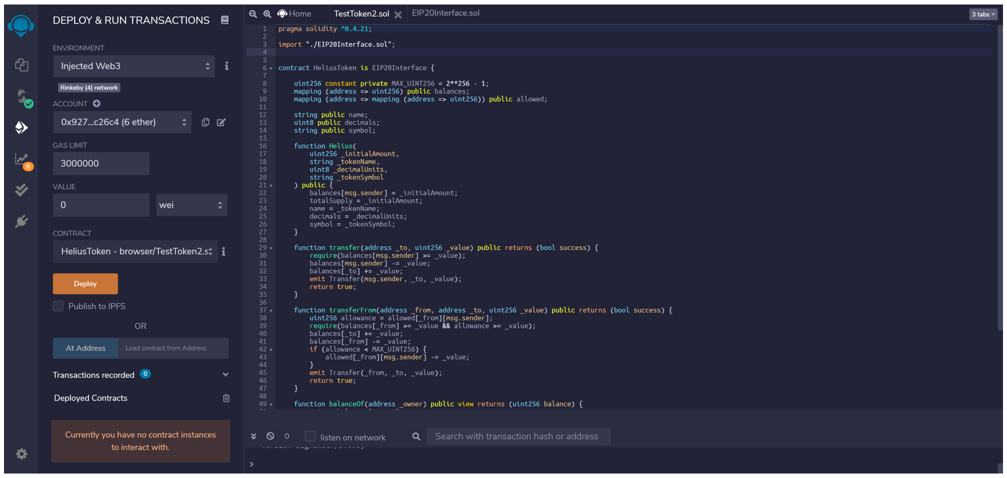
Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 15

February 28, 2025

Contents

1 Aim

2 Demo



ENVIRONMENT: Injected Web3
Rinkeby (4) network
ACCOUNT: 0x927...c26c4 (6 ether)
GAS LIMIT: 3000000
VALUE: 0 wei
CONTRACT: HeliusToken - browser/TestToken2.s

Currently you have no contract instances to interact with.

```
pragma solidity ^0.4.21;

import "./EIP20Interface.sol";


contract HeliusToken is EIP20Interface {

    uint256 constant private MAX_UINT256 = 2**256 - 1;
    mapping (address => uint256) public balances;
    mapping (address => mapping (address => uint256)) public allowed;

    string public name;
    uint8 public decimals;
    string public symbol;

    function Helius(
        uint256 _initialAmount,
        string _tokenName,
        uint8 _decimalUnits,
        string _tokenSymbol
    ) public {
        balances[msg.sender] = _initialAmount;
        totalSupply = _initialAmount;
        name = _tokenName;
        decimals = _decimalUnits;
        symbol = _tokenSymbol;
    }

    function transfer(address _to, uint256 _value) public returns (bool success) {
        require(balances[msg.sender] >= _value);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
        uint256 allowance = allowed[_from][msg.sender];
        require(balances[_from] >= _value && allowance >= _value);
        balances[_to] += _value;
        balances[_from] -= _value;
        if (allowance < MAX_UINT256) {
            allowed[_from][msg.sender] -= _value;
        }
        emit Transfer(_from, _to, _value);
        return true;
    }

    function balanceOf(address _owner) public view returns (uint256 balance) {
```

Figure 1: The Remix IDE

3 Frequently Asked Questions

3.1 What is the Concept of Gas in Solidity? How can a Developer Optimize Gas Usage in Smart Contracts?

Gas in Solidity refers to the computational cost required to execute operations on the Ethereum Virtual Machine (EVM). Each operation consumes a certain amount of gas, and users must pay for execution in Ether. Optimizing gas usage is crucial for reducing transaction costs and improving efficiency. Developers can optimize gas usage by:

- **Minimizing Storage Writes:** Using storage (e.g., `storage` variables) is expensive; prefer `memory` or `calldata` where possible.
- **Using Shorter Data Types:** Using `uint8` instead of `uint256` can save gas when appropriate.
- **Packing Storage Variables:** Grouping variables of the same type reduces storage slots used.
- **Avoiding Unnecessary Computation:** Precomputing values off-chain and passing them as function arguments reduces computation costs.
- **Using `view` and `pure` Functions:** These functions do not modify the blockchain and save gas.

3.2 What are the Key Features of Remix IDE and How Does it Simplify the Development of Ethereum Smart Contracts?

Remix IDE is a powerful web-based development environment for writing, testing, and deploying Solidity smart contracts. Key features include:

- **Built-in Solidity Compiler:** Allows real-time compilation with error highlighting.

- **Debugger:** Provides step-by-step execution for debugging smart contracts.
- **Deployment and Interaction:** Supports deployment on local, test, and mainnet environments.
- **Remix Plugins:** Extensible via plugins such as Solidity Static Analysis and Gas Estimator.
- **Integrated Terminal:** Allows running scripts, interacting with deployed contracts, and monitoring logs.

3.3 How Does the "Solidity Compiler" Plugin in Remix Work, and What Options Does it Provide for Developers?

The Solidity Compiler plugin in Remix is used to compile Solidity code into bytecode that the Ethereum Virtual Machine (EVM) can execute. It provides various options for developers:

- **Compiler Version Selection:** Allows choosing a specific Solidity version to ensure compatibility.
- **Optimization Settings:** Enables toggling optimizations to reduce gas costs.
- **EVM Target Selection:** Allows specifying the EVM version for backward compatibility.
- **Auto Compilation:** Automatically compiles the contract upon changes.
- **Metadata Generation:** Generates ABI (Application Binary Interface) and contract metadata.

3.4 4. What is the Difference Between Public, Internal, and Private Visibility in Solidity?

In Solidity, visibility modifiers determine access control for contract functions and state variables:

- **Public:** Accessible from within the contract, derived contracts, and externally via transactions.
- **Internal:** Accessible only within the contract and its derived contracts, but not externally.
- **Private:** Restricted to the defining contract; not accessible from derived contracts or externally.

4 Conclusion

Understanding gas optimization techniques, development tools like Remix IDE, and Solidity-specific features such as compiler options and visibility modifiers is crucial for writing efficient smart contracts. Optimizing gas usage can significantly reduce transaction costs, while utilizing tools like Remix simplifies the development and debugging process. Proper use of visibility modifiers ensures security and proper contract structure, improving maintainability and reliability of Ethereum smart contracts.

References

[1] Solidity Documentation. Available at: https://docs.soliditylang.org/

[2] Remix IDE Documentation. Available at: https://remix-ide.readthedocs.io/

[3] Ethereum Gas and Fees. Available at: https://ethereum.org/en/developers/docs/gas/

[4] Solidity Optimizations. Available at: https://docs.soliditylang.org/en/latest/internals/optimizations.html

[5] Solidity Visibility and Accessors. Available at: https://docs.soliditylang.org/en/latest/contracts.html#visibility-and-getters