Cyberscope

Audit Report

Libera Global AI

August 2024

Files 1_Storage.sol, 2_Owner.sol, 3_Ballot.sol, LiberaContract.sol, LiberaTokenERC20.sol, LiberaTokenIERC20.sol

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Audit Updates

Initial Audit	19 Aug 2024

Source Files

Filename	SHA256
LiberaTokenIERC20.sol	ec783469ab08b7c8ffcfcf07f588e64cf0c9075a465a007dc071b1cb2bd588c8
LiberaTokenERC20.sol	1fff493200c4a72331190526ef1dc7218db53dba51bb4ac665fb3d70aa2c4f2d
LiberaContract.sol	9f29b459274f2068cf2bcb7065b80b11afd00987411a512bff70c7519eb83c05
3_Ballot.sol	43fdc42cdd2085bafe1c6ca942a1ca6699d481c1eb5f68fb5a15b649b0827615
2_Owner.sol	9d19661d5f2ea63bd44db39cdfcfd0ac9219c3b33740e04b39345005c3a57ba4
1_Storage.sol	851044d0f208446ec8946fc2847f129058e240a821aed42972697d1d77d6d6ee

Overview

Storage Contract

The **Storage** contract is a simple smart contract designed to store and retrieve a single value on the blockchain. It provides basic functionality to save a `uint256` value using the `store` function and later retrieve it using the `retrieve` function. This contract serves as a fundamental example of how data can be persisted on the blockchain, allowing for a specific value to be stored and accessed by any user who interacts with the contract. Its primary purpose is to demonstrate the basic principles of state storage and retrieval within a smart contract environment.

Owner Contract

The **Owner** contract manages the ownership of a contract, providing mechanisms to control access to specific functions. Upon deployment, the contract assigns the deploying address as the initial owner. The contract includes functions to change ownership and retrieve the current owner's address, with the critical `changeOwner` function being restricted to the current owner only. This contract is designed to ensure that only the designated owner can execute sensitive operations, making it useful in scenarios where contract access control is required. The `Owner` contract is foundational in managing roles and permissions within more complex smart contracts, ensuring that key administrative functions are protected.

Ballot Contract

The **Ballot** contract is a decentralized voting system that allows participants to vote on a set of proposals. It includes mechanisms for vote delegation, enabling voters to transfer their voting power to others. The contract is initiated with a list of proposals, and a chairperson is designated with the authority to grant voting rights. Each voter has a weight, which can accumulate through delegation, and they can cast their vote on one of the proposals. The contract also provides functionality to compute the winning proposal based on the votes received and to return the name of the winning proposal. This contract is designed to

facilitate a secure and transparent voting process, ensuring that all votes are accounted for and that the results reflect the collective decision of the participants.

LiberaPacket Contract

The **LiberaPacket** contract is a sophisticated system for managing data packets and distributing rewards associated with them. The contract allows the owner to create and manage data packets, each identified by a unique data hash. These packets can be validated, and rewards can be set and distributed to the packet owners. The contract also includes safeguards, such as checks to prevent the creation of duplicate packets and ensuring that only validated packets receive rewards. The reward distribution is tightly controlled, with the reward pool wallet being the sole entity authorized to distribute rewards. This contract is designed to ensure a secure and organized process for managing data packets and the associated incentives, making it ideal for use cases where data integrity and fair reward distribution are paramount.

LiberaToken Contract (ERC20 Implementation)

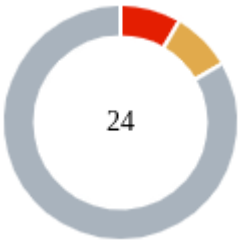
The **LiberaToken** contract is a standard ERC20 token implementation that allows for the creation and management of a cryptocurrency token named "LiberaToken" with the symbol "LIBE." Upon deployment, a fixed initial supply of tokens is minted and assigned to the contract deployer. The contract inherits from OpenZeppelin's ERC20 and Ownable contracts, providing a robust and secure implementation of the ERC20 standard with additional ownership controls. This contract enables basic token functionalities such as transferring tokens, checking balances, and managing allowances, making it suitable for use in various decentralized applications and token-based ecosystems.

LiberaTokenIERC20 Contract (Custom Implementation)

This **LiberaToken** contract provides a custom implementation of the ERC20 standard, focusing on direct control over token operations such as transfers, allowances, and balance inquiries. Similar to the previous LiberaToken contract, it manages the LiberaToken with the same name and symbol. However, this version includes custom error handling and additional constraints, such as preventing users from checking other users' balances unless they are the owner. It also includes detailed checks and custom error messages for various

operations, such as transfers and approvals, enhancing security and clarity within the contract.

Findings Breakdown



Critical	2
Medium	2
Minor / Informative	20

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	2	0	0	0
Minor / Informative	20	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	POV	Phishing Origin Vulnerability	Unresolved
●	UDV	Unrestricted Delegate Voting	Unresolved
●	MRD	Multiple Reward Distribution	Unresolved
●	ZAD	Zero Address Delegation	Unresolved
●	CFO	Code Function Optimization	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DCP	Debugging Code Presence	Unresolved
●	EHO	Error Handling Omission	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MTVH	Missing Tie Vote Handling	Unresolved
●	MU	Modifiers Usage	Unresolved

●	NBR	Non-Compliant Balance Restriction	Unresolved
●	PNC	Proposal Name Conflict	Unresolved
●	RCS	Redundant Comment Segment	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	USC	Unused Storage Contract	Unresolved
●	UVI	Unused Variable Initialization	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

POV - Phishing Origin Vulnerability

Criticality	Critical
Location	LiberaTokenIERC20.sol#L57,75,86,105,116
Status	Unresolved

Description

The contract uses the `tx.origin`, to perform security-related functionalities like user authentication and access control. The `tx.origin` is a global variable in Solidity that represents the original sender of the transaction. It refers to the external account that initiated the transaction and does not change even if the transaction is forwarded through multiple contracts.

However, using `tx.origin` for user authorization introduces a severe security vulnerability known as the "identity spoofing" attack. In this attack, a malicious contract can exploit the `tx.origin` vulnerability by impersonating the original sender of a transaction and bypassing user authorization checks. This can lead to unauthorized access, unauthorized actions on behalf of legitimate users, and potential financial losses or manipulation of contract behavior.

```
   function transfer(address recipient, uint256 amount)
external override returns (bool) {
        ...
        _balances[tx.origin] -= amount; //
_balances[msg.sender] -= amount;
        _balances[recipient] += amount;
        ...
        return true;
    }

    function approve(address spender, uint256 amount) external
override returns (bool) {
        ...
        _allowances[tx.origin][spender] = amount; //
_allowances[msg.sender][spender] = amount;
        ...

        return true;
    }


    function transferFrom(address sender, address recipient,
uint256 amount) external override returns (bool) {
        ...
        _balances[sender] -= amount;
        _balances[recipient] += amount;
        _allowances[sender][tx.origin] -= amount; //
_allowances[sender][msg.sender] -= amount;
    ...
    }

    function increaseAllowance(address spender, uint256
addedValue) public returns (bool) {
        ...
        _allowances[tx.origin][spender] += addedValue; //
_allowances[msg.sender][spender] += addedValue;
        ...

        return true;
    }

    function decreaseAllowance(address spender, uint256
subtractedValue) public returns (bool) {
        ...
        _allowances[tx.origin][spender] = currentAllowance -
subtractedValue; // _allowances[msg.sender][spender] =
currentAllowance - subtractedValue;
        ...
        return true;
    }
```

Recommendation

To mitigate the vulnerability associated with using `tx.origin` for user authorization, it is crucial to implement more secure user authentication and authorization mechanisms within the smart contract. It is recommended to use `msg.sender` instead. Replace all instances of `tx.origin` with `msg.sender` for user authentication and authorization. `msg.sender` represents the immediate sender of the call and is not susceptible to identity spoofing attacks.

UDV - Unrestricted Delegate Voting

Criticality	Critical
Location	3_Ballot.sol#L71
Status	Unresolved

Description

The contract is designed with a `delegate` function that allows users to delegate their vote to another user. However, this function can be invoked by any user, even if the `msg.sender` does not currently have the ability to vote (i.e., their voting weight is zero). As a result, the `voted` status of the user is set to true, preventing them from voting in the future if they later gain voting weight. This creates a scenario where a user, who could potentially acquire the right to vote in the future, loses their ability to vote because they invoked the delegate function prematurely.

```
   function delegate(address to) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");
        require(to != msg.sender, "Self-delegation is
disallowed.");

        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in
delegation.");
        }
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount +=
sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }
```

Recommendation

It is recommended to implement a check within the `delegate` function to ensure that the user invoking the delegation has a non-zero voting weight before allowing the delegation to proceed. This would prevent users without voting rights from prematurely setting their `voted` status to true, thus preserving their ability to vote if they gain the right in the future.

MRD - Multiple Reward Distribution

Criticality	Medium
Location	LiberaContract.sol#L108
Status	Unresolved

Description

The contract is designed with a `distributeReward` function that allows for the distribution of rewards based on a specific packet ID. However, the function does not include safeguards to prevent multiple executions for the same packet ID. As a result, if the function is invoked multiple times with the same ID, the contract will distribute rewards multiple times, leading to potential overpayment and depletion of the reward pool.

```
   function distributeReward(uint256 _id) external override
onlyRewardPool {
        if (packets[_id].validated == false)
            revert PacketIsNotValid();
        if (packets[_id].reward == 0)
            revert RewardShouldNotBeZero();

        // token.approve(rewardPoolWallet,
packets[_id].reward);
        // token.transferFrom(rewardPoolWallet,
packets[_id].owner, packets[_id].reward);

        token.transfer(packets[_id].owner,
packets[_id].reward);

        packets[_id].rewarded += packets[_id].reward;

        emit DistributeReward(_id, packets[_id].reward);
    }
```

Recommendation

It is recommended to implement checks within the `distributeReward` function to prevent multiple distributions for the same packet ID. This could include adding a flag or condition that ensures the function cannot be executed again for a packet once the rewards

have been distributed. This will protect against unintended multiple reward distributions and maintain the integrity of the reward system.

ZAD - Zero Address Delegation

Criticality	Medium
Location	3_Ballot.sol#L71
Status	Unresolved

Description

The contract is designed with a `delegate` function that allows users to delegate their vote to another address. However, the function lacks a check to prevent the zero address from being set as the `to` address for delegation. If the `to` address is set to the zero address, the delegate will also be zero, which could inadvertently increase the `weight` of the zero address. This could lead to unintended and potentially harmful consequences within the voting process.

```
   function delegate(address to) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");
        require(to != msg.sender, "Self-delegation is
disallowed.");

        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in
delegation.");
        }
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount +=
sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
```

```
    }
```

Recommendation

It is recommended to include a check within the `delegate` function to prevent the zero address from being passed as the `to` parameter. This will ensure that only valid addresses can be delegated votes, thereby maintaining the integrity of the voting process and preventing unintended manipulation of the proposal vote counts.

CFO - Code Function Optimization

Criticality	Minor / Informative
Location	LiberaTokenIERC20.sol#L57,71,75,86,105,116
Status	Unresolved

Description

The contract contains several functions and code segments that could benefit from optimization. Specifically, the `transfer` and `transferFrom` functions contain similar code segments that could be consolidated to reduce redundancy. Additionally, the `increaseAllowance` and `decreaseAllowance` functions do not utilize the existing `approve` function, leading to potential code duplication and increased maintenance complexity.

```
  function transfer(address recipient, uint256 amount) external
override returns (bool) {
        if (recipient == address(0))
            revert TransferToTheZeroAddress();
        if(_balances[tx.origin] < amount)
            revert TransferAmountExceedsBalance();

        _balances[tx.origin] -= amount; // _balances[msg.sender] -=
amount;
        _balances[recipient] += amount;

        emit Transfer(tx.origin, recipient, amount);

        return true;
    }

    function allowance(address owner, address spender) external view
override returns (uint256) {
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 amount) external override
returns (bool) {
        if(spender == address(0))
            revert ApproveToTheZeroAddress();

        _allowances[tx.origin][spender] = amount; //
_allowances[msg.sender][spender] = amount;

        emit Approval(tx.origin, spender, amount); // emit
Approval(msg.sender, spender, amount);

        return true;
    }

    function transferFrom(address sender, address recipient, uint256
amount) external override returns (bool) {
        if (sender == address(0))
            revert TransferFromTheZeroAddress();
        if (recipient == address(0))
            revert TransferToTheZeroAddress();
        if(_balances[sender] < amount)
            revert TransferAmountExceedsBalance();
        if (_allowances[sender][tx.origin] < amount)
            revert TransferAmountExceedsAllowance();

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        _allowances[sender][tx.origin] -= amount; //
_allowances[sender][msg.sender] -= amount;
```

```
        emit Transfer(sender, recipient, amount);

        return true;
    }

    function increaseAllowance(address spender, uint256 addedValue)
public returns (bool) {
        if (spender == address(0))
            revert IncreaseAllowanceToTheZeroAddress();

        _allowances[tx.origin][spender] += addedValue; //
_allowances[msg.sender][spender] += addedValue;

        emit Approval(tx.origin, spender,
_allowances[tx.origin][spender]); // emit Approval(msg.sender, spender,
_allowances[msg.sender][spender]);

        return true;
    }

    function decreaseAllowance(address spender, uint256
subtractedValue) public returns (bool) {
        if (spender == address(0))
            revert DecreaseAllowanceToTheZeroAddress();

        uint256 currentAllowance = _allowances[tx.origin][spender]; //
uint256 currentAllowance = _allowances[msg.sender][spender];

        if (currentAllowance < subtractedValue)
            revert DecreasedAllowanceBelowZero();

        _allowances[tx.origin][spender] = currentAllowance -
subtractedValue; // _allowances[msg.sender][spender] = currentAllowance
- subtractedValue;

        emit Approval(tx.origin, spender,
_allowances[tx.origin][spender]); // emit Approval(msg.sender, spender,
_allowances[msg.sender][spender]);

        return true;
    }
```

Recommendation

It is recommended to implement an internal function that encapsulates the shared logic of the `transfer` and `transferFrom` functions, allowing both to call this function and thereby reduce code duplication. Similarly, the `increaseAllowance` and

`decreaseAllowance` functions should be refactored to utilize the `approve` function, ensuring consistency across the contract and enhancing code maintainability. This approach will optimize the contract, making it more efficient and easier to manage.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	LiberaContract.sol#L62,82,95
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically, the owner has the ability to set the characteristics of data packets, determine the rewards associated with them, and validate or invalidate these packets.

```
function setPacket(address _owner, string calldata _name, string
calldata _dataHash, string calldata _dataType) external override
onlyOwner {
        ...
        dataHashToPacketId[_dataHash] = nextPacketId;
        nextPacketId++;
    }

    function validatePacket(uint256 _id, bool _isValid) external
override onlyOwner {
        if (packets[_id].id != _id)
            revert PacketDoesNotExist();

        packets[_id].validated = _isValid;

        emit ValidatePacket(_id, _isValid);

        if(!_isValid) {
            packets[_id].reward = 0;
        }
    }

    function setReward(uint256 _id, uint256 _reward) external
override onlyOwner {
        if (_reward == 0)
            revert RewardShouldNotBeZero();
        if (packets[_id].id != _id)
            revert PacketDoesNotExist();
        if (packets[_id].validated == false)
            revert PacketIsNotValid();

        packets[_id].reward = _reward;

        emit SetReward(_id, _reward);
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DCP - Debugging Code Presence

Criticality	Minor / Informative
Location	2_Owner.sol#L5,33
Status	Unresolved

Description

The contract includes the `import` statement of `console.sol` of hardhat and utilizes the `console.log` function for debugging purposes. While debugging statements like these are useful during the development and testing phases, they serve no purpose in a deployed contract and result in unnecessary gas costs. The inclusion of debugging code in the final deployed contract can lead to increased transaction fees and overall inefficiency.

```
import "hardhat/console.sol";
...
console.log("Owner contract deployed by:", msg.sender);
```

Recommendation

It is recommended to remove all debugging code, including the import statement and any `console.log` statements, from the contract before deployment. This will ensure that the contract is optimized for gas efficiency and does not include any extraneous code that could increase transaction costs.

EHO - Error Handling Omission

Criticality	Minor / Informative
Location	3_Ballot.sol#L100
Status	Unresolved

Description

The contract uses the `vote` function, which takes a `uint` as a parameter representing the proposal number. Although the function will automatically revert if the provided proposal number is out of the range of the proposals array, the code does not provide a clear and descriptive error message to indicate the specific issue. This lack of clarity in error messaging could lead to confusion for users and developers when diagnosing transaction failures, potentially increasing debugging time and reducing the overall usability of the smart contract.

```
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If 'proposal' is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }
```

Recommendation

It is recommended to add a specific check within the `vote` function to validate that the proposal number is within the bounds of the proposals array. This check should include a detailed error message to inform users when the proposal number is invalid, thus enhancing the clarity of error messages and improving the contract's overall robustness.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	LiberaTokenIERC20.sol#L39,40 LiberaContract.sol#L51,53 3_Ballot.sol#L36
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
decimals
_totalSupply
rewardPoolWallet
deployTime
chairperson
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MMI - Mapping Misuse Issue

Criticality	Minor / Informative
Location	3_Ballot.sol#L110
Status	Unresolved

Description

The contract is currently using the `proposals[proposal].voteCount += sender.weight;` statement in a manner that suggests it operates as a mapping rather than an array. This can lead to unintended behavior in the voting process, as the indexing of proposals does not align with how mappings are accessed. Since mappings do not inherently track an order or a range of keys, relying on them as if they were arrays can result in misallocation of votes or potential vulnerabilities in tallying the correct vote counts.

```
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        ...
        // If 'proposal' is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }
```

Recommendation

It is recommended to ensure that `proposals[proposal]` is correctly used as an array and that the indexing corresponds accurately to the intended proposal. If the intention was to use a mapping, the contract logic should be revised to reflect this, ensuring proper key-value access and safeguarding against any potential misuse that could compromise the integrity of the voting process.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	3_Ballot.sol#L63
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(voters[voter].weight == 0)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	3_Ballot.sol#L54,71,100
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
    function giveRightToVote(address voter) public {
        ...
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    function delegate(address to) public {
    ...
    }

    function vote(uint proposal) public {
    ...
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MTVH - Missing Tie Vote Handling

Criticality	Minor / Informative
Location	3_Ballot.sol#L117
Status	Unresolved

Description

The contract is designed to compute the winning proposal using the `winningProposal` function, which takes into account all previous votes. However, if two or more proposals receive the same maximum number of votes, only the first proposal encountered with the maximum vote count will be counted as the winner. This creates a scenario where the contract does not properly handle tie situations, potentially leading to unfair or unintended outcomes.

```
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }
```

Recommendation

It is recommended to modify the `winningProposal` function to handle cases where multiple proposals have the same highest vote count. This could involve implementing additional logic to either break ties in a fair manner or recognize multiple winning proposals, depending on the desired outcome of the voting process. This change will ensure that the contract operates fairly and as intended, even in tie scenarios.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	LiberaContract.sol#L83,98,100,109 3_Ballot.sol#L73,103
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
if (packets[_id].id != _id)
    revert PacketDoesNotExist();
...
if (packets[_id].validated == false)
  revert PacketIsNotValid();
```

```
require(!sender.voted, "You already voted.");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

NBR - Non-Compliant Balance Restriction

Criticality	Minor / Informative
Location	LiberaTokenIERC20.sol#L33,50
Status	Unresolved

Description

The contract is designed to restrict users from retrieving the token balances of other addresses, only allowing them to view their own balance. However, this approach does not comply with the ERC20 standard, which mandates that the `balanceOf` function should be publicly accessible for any address. Additionally, this restriction is ultimately ineffective, as the balances of all users can still be accessed by directly reading the contract's storage slots, undermining the intended privacy.

```
    mapping(address => uint256) private _balances;

    function balanceOf(address account) external view override
returns (uint256) {
        if (tx.origin != owner() && account != tx.origin)
            revert YouCanOnlyCheckTheBalanceOfYourWallet();

        return _balances[account];
    }
```

Recommendation

It is recommended to remove the restrictive logic from the `balanceOf` function and ensure full compliance with the ERC20 standard, allowing any address to query the balances of others. This change will maintain standard functionality, improve transparency, and align the contract with widely accepted token practices.

PNC - Proposal Name Conflict

Criticality	Minor / Informative
Location	3_Ballot.sol#L117,133
Status	Unresolved

Description

The contract is currently designed to store proposals by their names, which can lead to ambiguity when multiple proposals share the same name. Specifically, the `winnerName` function only returns the name of the winning proposal based on vote count but does not differentiate between proposals with identical names. This can cause confusion and errors in determining the actual winning proposal, as the function does not provide any mechanism to distinguish between proposals with duplicate names.

```
        for (uint i = 0; i < proposalNames.length; i++) {
            // 'Proposal({...})' creates a temporary
            // Proposal object and 'proposals.push(...)'
            // appends it to the end of 'proposals'.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
        ...
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
```

Recommendation

It is recommended to implement a unique identifier for each proposal, in addition to the name, to ensure that the `winnerName` function can accurately return the specific winning proposal. This could involve assigning a unique ID to each proposal upon creation and adjusting the logic to return this ID alongside the proposal name, thereby eliminating ambiguity and ensuring accurate results.

RCS - Redundant Comment Segment

Criticality	Minor / Informative
Location	LiberaContract.sol#L114 LiberaTokenIERC20.sol#L63,79,98,109,125
Status	Unresolved

Description

The contract contains a commented-out segment within the code. This does not reflect any actual implementation in the contract since they are commented-out. Commented-out code can be confusing, as it may not be clear why the code was commented out or whether it is intended for future use.

```
   function distributeReward(uint256 _id) external override
onlyRewardPool {
        ...
        // token.approve(rewardPoolWallet,
packets[_id].reward);
        // token.transferFrom(rewardPoolWallet,
packets[_id].owner, packets[_id].reward);
        }
```

```
_balances[tx.origin] -= amount; // _balances[msg.sender] -=
amount;
...
_allowances[tx.origin][spender] = amount; //
_allowances[msg.sender][spender] = amount;
emit Approval(tx.origin, spender, amount); // emit
Approval(msg.sender, spender, amount);
...
_allowances[sender][tx.origin] -= amount; //
_allowances[sender][msg.sender] -= amount;
...
_allowances[tx.origin][spender] += addedValue; //
_allowances[msg.sender][spender] += addedValue;
emit Approval(tx.origin, spender,
_allowances[tx.origin][spender]); // emit Approval(msg.sender,
spender, _allowances[msg.sender][spender]);
...
_allowances[tx.origin][spender] = currentAllowance -
subtractedValue; // _allowances[msg.sender][spender] =
currentAllowance - subtractedValue;
emit Approval(tx.origin, spender,
_allowances[tx.origin][spender]); // emit Approval(msg.sender,
spender, _allowances[msg.sender][spender]);
```

Recommendation

The team is advised to either remove the commented-out code segment from the code or update it to reflect its actual purpose and implementation. This will improve the code's readability and maintainability, reducing the potential for confusion or errors.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	LiberaContract.sol#L103
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
   function setReward(uint256 _id, uint256 _reward) external
override onlyOwner {
        ...
        packets[_id].reward = _reward;
    }
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

USC - Unused Storage Contract

Criticality	Minor / Informative
Location	1_Storage.sol#L18,26
Status	Unresolved

Description

The contract is designed to store and retrieve a number using the provided `store` and `retrieve` functions. However, this functionality is not utilized or referenced anywhere else within the codebase, resulting in a lack of meaningful utility for the contract. The presence of unused functions may indicate incomplete implementation or a lack of necessity for this functionality in the current scope of the project. This can lead to confusion, increase maintenance overhead, and potential security risks if future modifications are not carefully managed.

```
contract Storage {

    uint256 number;

    function store(uint256 num) public {
        number = num;
    }

    function retrieve() public view returns (uint256){
        return number;
    }
}
```

Recommendation

It is recommended to reconsider the actual functionality of the contract. If the storage and retrieval of a number are intended to serve a purpose within the system, the code should be revised to ensure that this functionality is properly integrated and utilized. Alternatively, if this feature is deemed unnecessary, consider removing the associated code to streamline the contract and reduce potential maintenance and security issues.

UVI - Unused Variable Initialization

Criticality	Minor / Informative
Location	LiberaContract.sol#L62
Status	Unresolved

Description

The contract contains a `setPacket` function that initializes several variables, including `name` and `dataType`, to set the characteristics of a packet. However, these variables are not utilized elsewhere in the contract, rendering them redundant. This unnecessary initialization increases gas consumption without providing any functional benefit, leading to inefficiencies in the contract.

```
   function setPacket(address _owner, string calldata _name,
string calldata _dataHash, string calldata _dataType) external
override onlyOwner {
        if (dataHashToPacketId[_dataHash] != 0)
            revert PacketDoesExist();

        packets[nextPacketId] = packetStructure(nextPacketId,
_owner, _name, _dataHash, _dataType, block.timestamp, false, 0,
0);

        emit SetPacket(_owner, _dataHash);

        dataHashToPacketId[_dataHash] = nextPacketId;
        nextPacketId++;
    }
```

Recommendation

It is recommended to reconsider the variables required for the initialization of packets. If these variables, are not intended to be used within the contract or in future implementations, they could be removed to reduce gas consumption and streamline the contract's code.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	LiberaContract.sol#L9,62,74,82,95,108
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
struct packetStructure {
        uint256 id;
        address owner;
        string name;
        string dataHash;
        string dataType;
        uint256 timestamp;
        bool validated;
        uint256 reward;
        uint256 rewarded;
    }
string calldata _name
string calldata _dataHash
string calldata _dataType

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	LiberaContract.sol#L100,109
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
packets[_id].validated == false
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	LiberaContract.sol#L512_Owner.sol#L44
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
rewardPoolWallet = _rewardPoolWallet
owner = newOwner
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	LiberaTokenIERC20.sol#L2 LiberaTokenERC20.sol#L2 LiberaContract.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.4;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	LiberaContract.sol#L117
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
token.transfer(packets[_id].owner, packets[_id].reward)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

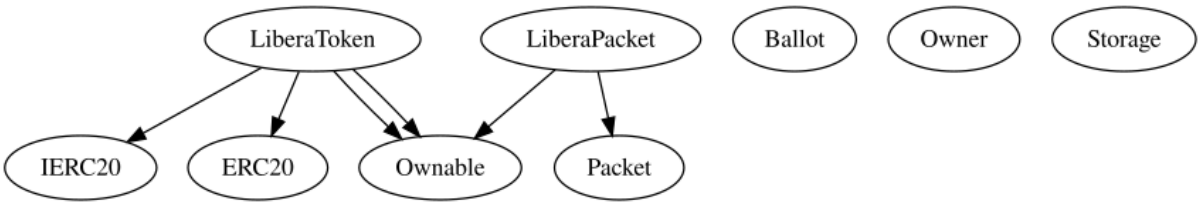
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
LiberaToken	Implementation	IERC20, Ownable		
		Public	✓	Ownable
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
LiberaToken	Implementation	ERC20, Ownable		
		Public	✓	ERC20 Ownable
Packet	Interface			
	setPacket	External	✓	-
	getPacketByDataHash	External		-
	validatePacket	External	✓	-
	setReward	External	✓	-

	distributeReward	External	✓	-
LiberaPacket	Implementation	Packet, Ownable		
		Public	✓	Ownable
	setPacket	External	✓	onlyOwner
	getPacketByDataHash	External		-
	validatePacket	External	✓	onlyOwner
	setReward	External	✓	onlyOwner
	distributeReward	External	✓	onlyRewardPool
Ballot	Implementation			
		Public	✓	-
	giveRightToVote	Public	✓	-
	delegate	Public	✓	-
	vote	Public	✓	-
	winningProposal	Public		-
	winnerName	Public		-
Owner	Implementation			
		Public	✓	-
	changeOwner	Public	✓	isOwner
	getOwner	External		-
Storage	Implementation			

	store	Public	✓	-
	retrieve	Public		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Storage
store
retrieve
Owner
isOwner
<Constructor>
changeOwner
getOwner
Ballot
<Constructor>
giveRightToVote
delegate
vote
winnerName
winningProposal
Packet (iface)
setPacket
getPacketByDataHash
validatePacket
setReward
distributeReward
LiberaPacket
<Constructor>
onlyRewardPool
OnlyRewardPool
setPacket
PacketDoesExist
getPacketByDataHash
validatePacket
PacketDoesNotExist
setReward
PacketIsNotValid
distributeReward
RewardShouldNotBeZero
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
LiberaToken
_mint
<Constructor>
decimals
totalSupply
YouCanOnlyCheckTheBalanceOfYourWallet
balanceOf
owner
transfer
Transfer
allowance
TransferToTheZeroAddress
transferFrom
TransferAmountExceedsBalance
TransferFromTheZeroAddress
TransferAmountExceedsAllowance
approve
ApproveToTheZeroAddress
IncreaseAllowanceToTheZeroAddress
increaseAllowance
Approval
decreaseAllowance
DecreaseAllowanceToTheZeroAddress
DecreasedAllowanceBelowZero

Summary

The LiberaToken contracts implement a standard ERC20 token mechanism with custom enhancements for ownership control, voting, reward distribution, and secure token management. This audit investigates potential security vulnerabilities, evaluates business logic consistency, and suggests improvements to optimize functionality and ensure robustness in these operations.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io