Security Audit

Date : 2024-03-06

Staking Upgradable Contracts

Executive Summary

Type	Smart Contract Audit
Audit Timeline	4 days
Runtime Environment	EVM
Languages	Solidity

Scope

0x5d8eD50c98FF5d1deB52E8a4Df668AB2d2324971

0x2bfBA3e1fc3Df9fdf8F5Bf87C70D5aDfE9B903C6

Summary of Findings

ID	Name	Description	Severity
H-01	Improper Initialization of Unlock Time in postConstruct	Unsafe Unlock Time Calculation	High
M-01	Incorrect Validation in updateUnlockTime Function	Inaccurate Validation Check for Unlock Time Update	Medium
L-01	Redundant Use of SafeMathUpgradeable with Solidity ^0.8.x	Unnecessary Use of SafeMathUpgradeable	Low
L-02	Lack of Input Validation for External Calls	Potential for Unintended Token Transfers	Low
L-03	] Inadequate Validation in Constructor of ERC1967Proxy	Lack of Contract Validation for Initial Logic Address	Low
L-04	Lack of Upgrade Authentication in ERC1967Upgrade	Unauthorized Upgrade Execution	Low
G-01	Gas Optimization via Caching msg.sender	Inefficient Gas Usage due to Repeated msg.sender Access	Gas
I-01	Misleading Event Parameter Comments	Incorrect names on Event Parameters	Informational

ID	Name	Description	Severity
H-01	Improper Initialization of Unlock Time in postConstruct	Unsafe Unlock Time Calculation	High
M-01	Incorrect Validation in updateUnlockTime Function	Inaccurate Validation Check for Unlock Time Update	Medium
L-01	Redundant Use of SafeMathUpgradeable with Solidity ^0.8.x	Unnecessary Use of SafeMathUpgradeable	Low
L-02	Lack of Input Validation for External Calls	Potential for Unintended Token Transfers	Low
L-03	] Inadequate Validation in Constructor of ERC1967Proxy	Lack of Contract Validation for Initial Logic Address	Low
L-04	Lack of Upgrade Authentication in ERC1967Upgrade	Unauthorized Upgrade Execution	Low
G-01	Gas Optimization via Caching msg.sender	Inefficient Gas Usage due to Repeated msg.sender Access	Gas
I-02	Token Recovery Function Risks	Potential Risks in Token Recovery Mechanism	Informational

Findings

[H-01] Improper Initialization of Unlock Time in postConstruct

Severity: Medium

Vulnerability Title: Unsafe Unlock Time Calculation

Description: The calculation of **unlockTime** in **postConstruct** directly adds **_lockupDays** to block.timestamp without converting days into seconds, potentially leading to incorrect lockup periods.

Impact
Misconfiguration of the unlock time could either unintentionally shorten or extend the staking period, affecting user expectations and contract functionality.

Recommendation
Modify the unlock time calculation by ensuring **_lockupDays** is properly converted into seconds **(_lockupDays * 1 days)**.

Proof of Concept

```
    function postConstruct(IERC20Upgradeable _token, uint256 _lockupDays)
public virtual initializer {
        // stake token should be valid address
        require(address(_token) != address(0x0), "staking token is
invalid");
        // validate supplied unlock time
        require(_lockupDays != 0 && _lockupDays <= 252 days, "invalid
lockupDays");

        // initialize state variable
        token = _token;
        unlockTime = block.timestamp + _lockupDays;
        // init dependent contracts
        __Ownable_init();
        __Pausable_init_unchained();
    }
```

[M-01] Incorrect Validation in updateUnlockTime Function

Severity: Medium

Vulnerability Title: Inaccurate Validation Check for Unlock Time Update

Description
The validation logic in the updateUnlockTime function incorrectly checks the new unlock time (_unlockTime) to be strictly less than block.timestamp + 252 days, which can inadvertently prevent setting an unlock time exactly at the 252-day limit. The comment // @audit unlockTime <= block.timestamp + 252 days, this should be the correct check correctly identifies the issue, suggesting that the validation should allow _unlockTime to be less than or equal to block.timestamp + 252 days.

Impact
This issue restricts the contract owner's ability to set the unlock time to its maximum intended limit, potentially reducing the flexibility needed for specific staking period adjustments.

Recommendation
Adjust the conditional check in the require statement to allow _unlockTime to be equal to block.timestamp + 252 days by changing the comparison operator from < to <=.

```
function updateUnlockTime(uint256 _unlockTime) external onlyOwner {
    // Ensure the new unlock time is no more than 252 days in the future
    require(_unlockTime > block.timestamp && _unlockTime <=
block.timestamp + 252 days, "invalid unlockTime");

    unlockTime = _unlockTime;

    emit UnlockTimeUpdated(msg.sender, _unlockTime);
}
```

[L-01] Redundant Use of SafeMathUpgradeable with Solidity ^0.8.x

Severity: Low Risk

Vulnerability Title: Unnecessary Use of SafeMathUpgradeable

Description
The contract uses SafeMathUpgradeable for arithmetic operations, which is redundant in Solidity ^0.8.x since the compiler automatically includes overflow and underflow checks.

Impact
While not a direct security risk, the unnecessary use of SafeMathUpgradeable increases gas costs for contract deployment and execution.

Recommendation
Remove SafeMathUpgradeable and rely on Solidity's built-in arithmetic checks to reduce gas costs.

[L-02] Lack of Input Validation for External Calls

Severity: Low Risk

Vulnerability Title: Potential for Unintended Token Transfers

Location: recoverERC20 function.

Description
The recoverERC20 function allows the contract owner to recover any ERC20 tokens sent to the contract accidentally. However, there's no explicit check for the null address (address(0)) before performing the token transfer. While there's a check to ensure the token being recovered isn't the staked token (require(address(_token) !=

address(token), "cannot withdraw the staking token");), not checking for the null address could lead to unintended behavior or gas wastage if the function is called with the null address as the _token parameter.

Recommendation
Implement an additional check to ensure _token is not the null address before proceeding with the recovery logic:

require(address(_token) != address(0), "Token address cannot be the zero address");

[L-03] Inadequate Validation in Constructor of ERC1967Proxy

Severity: Low Risk

Vulnerability Title: Lack of Contract Validation for Initial Logic Address

Location: ERC1967Proxy constructor.

Description
The constructor of ERC1967Proxy accepts an initial implementation logic address (_logic) and performs a delegate call with optional initialization data (_data). However, there's no explicit validation within the constructor to verify that the _logic address provided is a contract. This absence of validation could lead to setting an EOA (Externally Owned Account) or an invalid address as the proxy's implementation, which would render the proxy non-functional.

Recommendation
Implement a check within the constructor to ensure the _logic address is a contract before setting it as the implementation. This can be achieved using the Address.isContract function from OpenZeppelin's Address library:

require(Address.isContract(_logic), "ERC1967Proxy: new implementation is not a contract");

[L-04] Lack of Upgrade Authentication in ERC1967Upgrade

Severity: Low Risk

Vulnerability Title: Unauthorized Upgrade Execution

Location: ERC1967Upgrade functions _upgradeTo, _upgradeToAndCall, and _upgradeToAndCallSecure.

Description
The upgrade functions allow changing the contract's implementation address. The code snippets do not demonstrate any access control mechanism to restrict who can execute these upgrades. Without proper access control, any user could potentially trigger an upgrade and change the implementation to an arbitrary contract, leading to potential loss of funds or unexpected contract behavior. Even though the function is internal , adding additional checks will prevent it from getting called mistakenly from any derived contracts

Recommendation
Ensure that upgrade functions can only be called by authorized entities. This is typically handled by integrating access control mechanisms, such as OpenZeppelin's Ownable or AccessControl contracts, and protecting sensitive functions with modifiers like onlyOwner or role-based checks:

```
function _upgradeTo(address newImplementation) internal onlyOwner {
    ...
}
```

[G-01] Gas Optimization via Caching msg.sender

Severity: Gas

Vulnerability Title: Inefficient Gas Usage due to Repeated msg.sender Access

Location
Functions throughout the contract, specifically stake and withdraw.

Description
The contract's stake and withdraw functions access the global variable msg.sender multiple times. Each access of msg.sender consumes more gas than necessary when the value could be cached in a local variable at the start of the function. Solidity requires a certain amount of gas to access global variables, and repeated access to the same global variable within a single function execution can lead to inefficiencies in gas usage.

Impact
While not impacting the contract's security, this inefficiency leads to higher transaction costs for users. In high-usage contracts, especially on networks with high gas prices, optimizing for gas can significantly enhance user experience and reduce costs.
Reduce the gas footprint of your transactions, which can lead to cost savings for users.

Recommendation
Cache msg.sender in a local variable at the beginning of functions that reference it multiple times. This change minimizes the gas cost associated with repeated global variable access, optimizing the contract's overall gas usage.

Proof of Concept

```
function stake(uint256 amount) external override whenNotPaused {
    address sender = msg.sender; // Cache msg.sender
    require(amount > 0, "cannot stake 0");
    token.safeTransferFrom(sender, address(this), amount); // Use cached
sender

    if(_balances[sender] == 0) {
        totalNoOfStakers++;
    }
    _balances[sender] = _balances[sender].add(amount);
    totalSupply = totalSupply.add(amount);
    emit Staked(sender, amount, block.timestamp);
}

function withdraw(uint256 amount) public override {
    address sender = msg.sender; // Cache msg.sender
    require(amount > 0, "cannot withdraw 0");
    require(_balances[sender] >= amount, "bad withdraw");
    require(block.timestamp >= unlockTime, "withdraw is locked");

    _balances[sender] = _balances[sender].sub(amount);
    totalSupply = totalSupply.sub(amount);

    if(_balances[sender] == 0) {
        totalNoOfStakers--;
    }
    token.safeTransfer(sender, amount);

    emit Withdrawn(sender, amount, block.timestamp);
}
```

[I-01] Misleading Event Parameter Comments

Severity: Informational

Vulnerability Title: Incorrect names on Event Parameters

Description
 The parameter name on the Staked and Withdrawn events suggest potential misunderstandings regarding the functionality, which could mislead developers or auditors. The parameter user is the address of the sender of the transaction.

Proof of Concept

event Staked(address indexed user, uint256 amount, uint256 time);
event Withdrawn(address indexed user, uint256 amount, uint256 time);

Impact
The comments may cause confusion but do not affect the contract's security or functionality.

Recommendation
Change the names of the parameters from user to sender

[I-02] Token Recovery Function Risks

Severity: Informational

Vulnerability Title: Potential Risks in Token Recovery Mechanism

Description
The recoverERC20 function allows the contract owner to recover ERC20 tokens sent to the contract by mistake, except the staking token. However, without strict controls, this could be misused or lead to unintended consequences.

Impact
Mismanagement or misuse of the token recovery feature could undermine trust in the contract's governance or lead to loss of tokens.

Recommendation
Implement additional safeguards, such as time locks or requiring multiple signatures for token recovery operations.