CPChain PoS: RewardManager Security Audit Report

DogScan Security Team

August 4th, 2024

Contents

DogScan Security Audit Report

Project	CPChain PoS
Contract File	`RewardManager.sol`
Source Path	src/core/pos/RewardManager.sol
Commit	`b098dffd4589081b8a9996972cc044be552e321a`
Audit Date	August 4th, 2024
Report Version	1.0

1. Executive Summary

We conducted a comprehensive security audit of the `RewardManager` contract. This contract is responsible for reward distribution in the CPChain PoS system, managing both operator and staker rewards.

The audit discovered one critical-severity vulnerability that allows stakeholders to repeatedly withdraw their rewards without any balance accounting, potentially draining the entire reward pool. This vulnerability poses a severe threat to the protocol's financial integrity.

Overall Risk Rating: Critical

We strongly recommend the project team immediately fix this critical vulnerability and conduct thorough testing before any production deployment.

2. Audit Scope

The audit scope covers the complete functionality of the `RewardManager` contract:

Contract Information:

- Contract Type: Reward Distribution Management Contract
- Main Functions: Operator and stakeholder reward distribution, reward claiming

Key Audit Areas:

- Reward distribution logic
- Reward claiming mechanisms

- Access control patterns
- State management and accounting integrity
- Fund security and reentrancy protection

3. Audit Methodology

This audit employed a multi-agent AI security analysis framework specifically designed for smart contract security assessment:

1. Specialized Analysis Modules:

- **Financial Security Expert**: Focuses on reward distribution logic and fund security
- **State Management Expert**: Reviews state variable consistency and accounting integrity
- **Access Control Expert**: Evaluates permission management and role control
- **Reentrancy Expert**: Examines potential reentrancy attack vectors
- **Code Quality Expert**: Evaluates code standards and best practices

2. Comprehensive Analysis:

- Static code analysis focused on fund flows
- Reward distribution mechanism integrity verification
- Potential attack scenario analysis
- Cross-validation with similar patterns

4. Findings Summary

ID	Title	Severity	Status
C-01	Double-Claim Vulnerability in Stakeholder Reward Distribution	**Critical**	**Pending Fix**

5. Detailed Findings

[C-01] Double-Claim Vulnerability in Stakeholder Reward Distribution

Severity: Critical

Description The `stakeHolderClaimReward` function allows unlimited repeat claims of the same reward amount. After successfully transferring ETH to a stakeholder, the function fails to reset the stakeholder's reward balance in `stakerRewards[chainBase][msg.sender]`. This allows an attacker to repeatedly call the function until the contract's entire ETH balance is drained.

Technical Details

```
function stakeHolderClaimReward(address chainBase) external returns (bool) {
    uint256 stakeHolderAmount = _stakeHolderAmount(msg.sender, chainBase);
    require(
        stakeHolderAmount > 0,
        "RewardManager operatorClaimReward: stake holder amount need more then
            zero"
    );
    require(
        address(this).balance >= stakeHolderAmount,
        "RewardManager operatorClaimReward: Reward Token balance insufficient"
    );

    emit StakeHolderClaimReward(msg.sender, chainBase, stakeHolderAmount);

    (bool success, ) = payable(msg.sender).call{value: stakeHolderAmount}("");

    // Critical Missing: stakerRewards[chainBase][msg.sender] = 0;
    return success;
}
```

Contrast with the correct implementation in operator reward claiming:

```
function operatorClaimReward() external returns (bool) {
    uint256 claimAmount = operatorRewards[msg.sender];
    require(claimAmount > 0, "operator claim amount need more then zero");
    require(address(this).balance >= claimAmount, "balance insufficient");

    operatorRewards[msg.sender] = 0; // Correct balance reset

    emit OperatorClaimReward(msg.sender, claimAmount);
    (bool success, ) = payable(msg.sender).call{value: claimAmount}("");
    return success;
}
```

Impact

- **Complete Fund Drainage Risk**: Attackers can repeatedly call the function until contract balance reaches zero
- **Protocol Solvency Threat**: A single attacker can multiply their reward amount indefinitely
- **Other User Fund Loss**: Legitimate stakers and operators cannot claim their rightful rewards
- **System Integrity Breakdown**: Complete destruction of reward distribution mechanism credibility

Attack Scenario

1. Attacker as a legitimate staker earns any amount of rewards (e.g., 1 ETH)
2. Calls `stakeHolderClaimReward` to claim 1 ETH
3. Since balance is not reset, attacker can immediately call the function again
4. Repeats steps 2-3 until contract balance is exhausted

Recommendation Implement the following fix immediately:

```
function stakeHolderClaimReward(address chainBase) external returns (bool) {
    uint256 stakeHolderAmount = _stakeHolderAmount(msg.sender, chainBase);
    require(
        stakeHolderAmount > 0,
        "RewardManager operatorClaimReward: stake holder amount need more then zero"
    );
    require(
        address(this).balance >= stakeHolderAmount,
        "RewardManager operatorClaimReward: Reward Token balance insufficient"
    );

    // Critical Fix: Reset balance before transfer
    stakerRewards[chainBase][msg.sender] = 0;

    emit StakeHolderClaimReward(msg.sender, chainBase, stakeHolderAmount);

    (bool success, ) = payable(msg.sender).call{value: stakeHolderAmount}("");
    require(success, "Transfer failed");

    return success;
}
```

Additional Recommendations:

1. **Implement Reentrancy Protection**: Although current `call` method limits gas, consider adding `nonReentrant` modifier
2. **Follow CEI Pattern**: Move state updates before external calls
3. **Comprehensive Testing**: Write specific test cases to verify the fix's effectiveness

6. Architecture and Design Assessment

Design Strengths

1. **Access Control**: Implements role-based access control with appropriate protection for critical functions
2. **Event Logging**: Proper event emissions facilitate off-chain monitoring and auditing
3. **Modular Design**: Reward distribution logic is separated from other system components

Key Architectural Issues

1. **Inconsistent State Management**: Stakeholder and operator reward claiming functions have inconsistent implementations
2. **Insufficient Code Review**: Critical security vulnerability indicates inadequate code review processes
3. **Insufficient Test Coverage**: Lack of integration tests to detect double-claim scenarios

Systemic Risk Assessment

This contract serves as the core component of reward distribution with the following characteristics:

1. **High-Value Target**: Manages all staker and operator reward funds
2. **Critical Vulnerability Impact**: A single vulnerability can lead to complete fund loss
3. **Trust Dependency**: Protocol credibility entirely depends on fair reward distribution

7. Conclusion

This security audit identified **one critical-severity vulnerability** in the `RewardManager` contract.

Key Findings Summary:

- **Double-Claim Vulnerability**: Stakeholders can claim the same reward unlimited times
- **Fund Security Threat**: Could lead to complete reward pool drainage
- **Protocol Integrity Risk**: Destroys reward distribution mechanism credibility

Overall Risk Rating: Critical

This vulnerability poses an immediate and severe threat to the protocol's financial security. Attackers can exploit this vulnerability to drain the entire contract balance, leading to protocol insolvency and severely damaging user trust and protocol reputation.

Emergency Fix Recommendations:

1. **Immediate Fix**: Add state reset in the `stakeHolderClaimReward` function
2. **Thorough Testing**: Implement comprehensive test suites to verify the fix's effectiveness
3. **Code Review**: Strengthen code review processes to prevent similar issues
4. **Monitoring Deployment**: Implement strict monitoring mechanisms after the fix

We strongly recommend the project team immediately cease using the current contract, fix the vulnerability, redeploy, and conduct thorough testing in the production environment.