DogScan Security Audit Report

Project	**MultiTokenStakingDiamond**
Chain	BSC Testnet (ID 97)
Contract Address	0xEAb5FceBb8B3a698332D0Bd7a713C838f9e8F9cC
Audit Date	2025-07-06
Report Version	1.0

Table of Contents

1. Executive Summary

We conducted a security audit of the MultiTokenStakingDiamond contract system (an implementation of the EIP-2535 Diamond standard). The audit results show that the system's overall security status is good, with no serious security vulnerabilities discovered.

During the audit process, we identified some low-risk design issues, primarily involving centralization risks and precision handling. While the protocol's lending mechanism exhibits centralized characteristics, these are acceptable governance risks within the current design framework. Additionally, there are minor precision loss issues when handling high-precision tokens, but the impact is extremely limited.

We recommend that the project team consider the architectural recommendations outlined in this report before mainnet deployment, to further enhance the protocol's decentralization and user experience.

2. Audit Scope

The audit scope covers the MultiTokenStakingDiamond contract and its related Facets and libraries. The main proxy contract is located at `0xEAb5FceBb8B3a698332D0Bd7a713C838f9e8F9cC`.

The following source files were considered within the scope of this analysis: - `contracts/diamond/Diamond.sol` - `contracts/diamond/LibDiamond.sol` - `contracts/diamond/interfaces/IDiamondCut.sol` - `contracts/diamond/interfaces/IDiamondLoupe.sol` - `contracts/diamond/facets/DiamondCutFacet.sol` - `contracts/diamond/facets/DiamondLoupeFacet.sol` - `contracts/staking/MultiTokenStakingDiamond.sol` - `contracts/staking/LibMultiTokenStakingStorage.sol` - `contracts/staking/facets/StackOperationFacet.sol` - `contracts/staking/facets/StackQueryFacet.sol` - `contracts/Constants.sol` - `contracts/Utils.sol`

3. Audit Methodology

This audit employed a multi-agent AI security analysis workflow. The process deploys multiple specialized AI agents to analyze smart contract code, design, and on-chain context. Each agent focuses on specific categories of vulnerabilities, such as reentrancy, access control, mathematical vulnerabilities, and economic model risks. A chief security strategist AI then collects and synthesizes the findings from these specialized agents, conducting a comprehensive review to identify systemic risks, prioritize findings by severity, and compile this final comprehensive report.

4. Findings Summary

ID	Title	Severity	Status
L-01	Centralization risk: Privileged borrower role design poses potential risks	**Low**	**For Optimization**
L-02	Precision loss during deposits may	**Low**	**For Optimization**

ID	Title	Severity	Status
	lead to minor user fund loss		

5. Detailed Findings

[L-01] Centralization risk: Privileged borrower role design poses potential risks (Low)

Severity: Low

Description

The system contains a `borrowers` mapping in `LibMultiTokenStakingStorage.sol` and a `borrow` function in `StackOperationFacet.sol`. This allows authorized addresses to withdraw underlying assets from staking pools. From a design perspective, this is a centralized governance feature that may impact the degree of decentralization.

While this design may be intentional within the current governance framework (e.g., for protocol liquidity management, yield strategies, etc.), it does increase the protocol’s degree of centralization. The initialization logic in `LibMultiTokenStakingStorage.sol` grants this role to the contract deployer, which is a common governance pattern.

Impact

This design increases the protocol’s centralization risk. While it may be acceptable within the current governance framework, it could affect user trust in the protocol’s degree of decentralization in the long term.

Code Snippet

```
// File: contracts/staking/facets/StackOperationFacet.sol

function borrow(
    uint256 _pid,
    uint256 _amount,
    address _borrowee,
    uint256 _positionId
) external onlyBorrower returns (uint256) {
    // ...
    // Update borrowed amount
    pool.totalBorrowed = pool.totalBorrowed + _amount;
    // ...
    // Transfer LP tokens to borrower
    IERC20(pool.token).safeTransfer(msg.sender, _amount);
    // ...
}
```

Recommendations

Consider the following optimization approaches to enhance the protocol's decentralization: 1. **Governance mechanism**: Transfer "borrower" privileges to a multi-signature wallet or decentralized governance contract. 2. **Transparency**: Clearly document the purpose and use cases of the borrowing mechanism. 3. **Constraint mechanisms**: Consider setting borrowing limits or time restrictions as constraints. 4. **Monitoring mechanisms**: Implement event logging and monitoring mechanisms to improve operational transparency.

[L-02] Precision loss during deposits may lead to minor user fund loss (Low)

Severity: Low

Description

The `normalizeTokenAmount` function in the `Utils.sol` library is used to uniformly convert token amounts of different precisions to 18-decimal precision for internal calculations. For tokens with native precision higher than 18 decimals, the function uses integer division `_amount / (10 ** (decimals - 18))`.

Integer division in Solidity truncates remainders. This means that if a user deposits a token amount that is not a multiple of the conversion factor, the remainder portion will be discarded. For example, with a 20-decimal precision token, if a user deposits 199 smallest units, after the `199 / 100` calculation, it's internally recorded as only 1 unit (equivalent to 100 smallest units), and the remaining 99 smallest units' value stays in the contract, causing minor user loss.

Impact

For users using high-precision tokens, each deposit may result in extremely minor fund loss. The loss amount is typically negligible, but theoretically exists.

Code Snippet

```
// File: contracts/Utils.sol
```

```
function normalizeTokenAmount(
    address _token,
    uint256 _amount
) public view returns (uint256) {
    uint256 decimals = getTokenDecimals(_token);
    if (decimals == 18) return _amount;
    // For decimals > 18 cases, integer division truncates
remainder, causing minimal precision loss
    if (decimals > 18) return _amount / (10 ** (decimals - 18));
    return _amount * (10 ** (18 - decimals));
}
```

Recommendations

Consider the following optimization approaches: 1. **Documentation**: Clearly document the handling of high-precision tokens and potential minor losses. 2. **High-precision math libraries**: Consider using high-precision math libraries to reduce rounding errors. 3. **User notifications**: Provide appropriate notifications to users using high-precision tokens in the frontend interface.

6. Architecture and Design Recommendations

In addition to the above findings, we propose the following architectural and design recommendations to improve the protocol's robustness and user experience:

1. **Strengthen Access Control and Permission Management**: The protocol currently relies on a centralized `owner` address to manage whitelists, add/remove borrowers, and other critical operations. This EIP-2535 Diamond architecture design inherently concentrates control. We recommend transferring `owner` privileges to a multi-signature wallet or a timelock governance contract to reduce single point of failure risks.

2. **Optimize `view` Function Robustness**: The `canUserUnstake` function in `StackQueryFacet.sol` does not check whether the pool's `totalShares` is zero when calculating `userAmount`. If a pool is newly created with no stakers (`totalShares` = 0), calling this function may fail due to division by zero errors. While this won't directly cause fund loss, it will affect the availability of frontend or other services that depend on this function. We recommend adding checks before division operations.

3. **Add Pagination Mechanisms for Traversal Functions**: The `getUserPools` and `getUserPoolsWithRewards` functions traverse all pools that a user participates in. If a user participates in a large number of pools, calling these functions may consume excessive gas.

We recommend adding pagination parameters (such as `offset` and `limit`) to these functions to ensure controllable gas costs per call.

4. **Data Source Documentation for View Functions**: Some view functions (such as `userRewards`, `canUserUnstake`) use `balanceOf` to get real-time balances. While view functions themselves pose no security risks, we recommend clearly documenting the data source characteristics of these functions in documentation so integrators can properly understand and use them.

7. Conclusion

This audit found that the MultiTokenStakingDiamond contract system has good overall security status. While there are some low-risk design issues and areas for optimization, none constitute serious security threats.

The protocol's Diamond architecture implementation complies with the EIP-2535 standard, with reasonable core business logic design. The identified issues mainly focus on the degree of centralization in governance mechanisms and detail optimization in precision handling. We assess the protocol's overall security status as **"Low Risk"**.

We recommend that the project team consider the architectural recommendations outlined in this report before mainnet launch, to further enhance the protocol's decentralization and user experience.

8. Disclaimer

This audit report is for reference only and does not constitute investment advice. The analysis is based on smart contract source code provided at a specific point in time and is not an endorsement of the project or a guarantee of its security. Smart contracts have inherent risks, including the possibility of undiscovered vulnerabilities. Users should conduct their own research and exercise extreme caution when interacting with any smart contract.