

## ## Smart Contract Audit Report - Memefund and Memestake

This report summarizes the findings of a comprehensive audit conducted on the provided Memefund and

### \*\*Vulnerabilities:\*\*

#### \*\*1. Reentrancy Vulnerability\*\*

\* \*\*Severity:\*\* High

\* \*\*Description:\*\* The `deposit` and `withdraw` functions in the Memestake contract call external functions.

\* \*\*Impact:\*\* An attacker could execute a reentrancy attack and drain the contract's funds. The financial impact is significant.

\* \*\*Mitigation:\*\*

- Implement a reentrancy guard pattern by using a `nonReentrant` modifier or a similar technique. This prevents recursive calls to the vulnerable functions.

#### \*\*2. Missing Error Handling\*\*

\* \*\*Severity:\*\* High

\* \*\*Description:\*\* Several functions in the Memestake and Memefund contracts lack proper error handling.

\* \*\*Impact:\*\* In case of a failed transaction, the code might proceed as if it succeeded, potentially leading to incorrect state or loss of funds.

\* \*\*Mitigation:\*\*

- Add appropriate error handling mechanisms in functions that interact with external contracts or perform critical operations.

#### \*\*3. Ownership Concentration\*\*

\* \*\*Severity:\*\* High

\* \*\*Description:\*\* The contract has a single owner with complete control over all functions. This centralizes power.

\* \*\*Impact:\*\* A malicious owner could manipulate the contract's functionalities, freeze funds, or potentially drain the contract.

\* \*\*Mitigation:\*\*

- Implement a multi-signature wallet or a governance system that requires multiple parties to approve critical actions.
- Consider implementing a timelock mechanism for significant changes to the contract's parameters, allowing for a delay and review process.

#### \*\*4. No Withdrawal Limits\*\*

\* \*\*Severity:\*\* Medium

\* \*\*Description:\*\* The `withdraw` function in the Memestake contract allows users to withdraw their entire balance.

\* \*\*Impact:\*\* Attackers could exploit this vulnerability to drain funds by manipulating the `withdraw` function.

\* \*\*Mitigation:\*\*

- Introduce a withdrawal limit for each user or implement a cooling-off period before allowing full withdrawal.

#### \*\*5. Centralization Risk\*\*

\* \*\*Severity:\*\* Medium

\* \*\*Description:\*\* The contract relies heavily on a central rebase oracle and a single owner, leading to a centralization risk.

\* \*\*Impact:\*\* An attacker could manipulate the rebase mechanism through the oracle, potentially causing significant financial loss.

\* \*\*Mitigation:\*\*

- Explore decentralized oracles or a mechanism for choosing the rebase oracle, ensuring that the system is not controlled by a single entity.
- Consider using a timelock mechanism for the owner's actions, allowing for a delay and review process before executing critical changes.

#### \*\*6. Potential Gas Limit Assumptions\*\*

\* \*\*Severity:\*\* Low

\* \*\*Description:\*\* The `getMultiplier` function in the Memestake contract assumes a fixed gas limit for every transaction. This could lead to gas limit errors in certain scenarios.

\* \*\*Impact:\*\* Errors in the reward calculation could potentially lead to inaccurate distribution of tokens, resulting in unfair rewards.

\* \*\*Mitigation:\*\*

- Conduct more thorough testing to verify the gas limit assumption in various scenarios and ensure accurate calculations.
- Consider implementing a gas limit estimation mechanism within the contract to dynamically adjust the gas limit based on transaction complexity.

## **\*\*7. Magic Numbers\*\***

\* \*\*Severity:\*\* Moderate

\* \*\*Description:\*\* The contract uses magic numbers like `1e18` for token decimals. While these are likely correct, using descriptive constants would improve readability and maintainability.

\* \*\*Impact:\*\* This could make the code harder to understand and maintain, potentially leading to errors if the values are not updated consistently.

\* \*\*Mitigation:\*\*

- Define these numerical values as constants at the top of the file for better readability and clarity.

## **\*\*8. Naming Conventions\*\***

\* \*\*Severity:\*\* Low

\* \*\*Description:\*\* Some variable names like `mFundReward` and `accMfundPerShare` could benefit from more descriptive naming to clearly indicate their purpose.

\* \*\*Impact:\*\* This might make the code slightly harder to understand, especially for developers unfamiliar with the project's internal naming.

\* \*\*Mitigation:\*\*

- Employ more descriptive and standard naming conventions to improve code clarity and maintainability.

## **\*\*9. Code Organization\*\***

\* \*\*Severity:\*\* Low

\* \*\*Description:\*\* The code's complexity could potentially benefit from modularization, splitting it into separate contracts or functions where appropriate.

\* \*\*Impact:\*\* This might make the code harder to maintain and scale as the project grows.

\* \*\*Mitigation:\*\*

- Consider modularizing the code into smaller, more manageable contracts to improve maintainability and scalability.

## **\*\*Recommendations:\*\***

\* \*\*Prioritize:\*\* Address the high-severity vulnerabilities, specifically reentrancy, missing error handling, and the lack of a withdrawal limit.

\* \*\*Implement:\*\* Implement a robust reentrancy guard mechanism to prevent potential attacks.

\* \*\*Enhance:\*\* Add appropriate error handling and fallback mechanisms throughout the contract.

\* \*\*Decentralize:\*\* Explore ways to decentralize control by introducing a multi-signature wallet or a governance system.

\* \*\*Limit:\*\* Implement withdrawal limits to prevent rapid drainage of funds.

\* \*\*Test Thoroughly:\*\* Conduct comprehensive testing, including reentrancy tests, to ensure the contract's security and reliability.

This report highlights several vulnerabilities and potential risks within the Memefund and Memestake contracts. It is crucial to address these issues promptly to ensure the security and integrity of the project.