# Security Audit Report for MultiSig.sol

**Audit Conducted by**: MahdiFa
**Date**: September 23, 2025
**Contract**: MultiSig.sol
**Repository**: MultiSig-Wallet

## Table of Contents



## 1. Executive Summary

This report presents the findings of a security audit conducted on the `MultiSig.sol` smart contract, a multi-signature wallet designed to manage transactions with multiple owner confirmations and a time delay mechanism. The audit identified **two security vulnerabilities** and **three**

**optimization/informational issues**. The high-severity issue involves a reentrancy vulnerability that could lead to the draining of the contract's funds. The medium-severity issue allows bypassing the transaction time delay, undermining the contract’s security design. Additionally, one low-severity gas optimization and two informational issues were identified to improve code robustness and maintainability.

The audit was performed using a combination of manual code review and logical analysis, focusing on security, gas efficiency, and best practices. Recommendations are provided to address all findings, ensuring the contract's security and performance.

## 2. Scope of Audit

The audit focused on the `MultiSig.sol` smart contract, which implements a multi-signature wallet with the following key functionalities:

- Proposing transactions by owners.
- Confirming transactions by multiple owners.
- Executing transactions after a specified time delay.
- Revoking confirmations by owners.
- Receiving Ether deposits.

**Files Audited**:

- MultiSig.sol

**Key Parameters**:

- Solidity version: ^0.8.18
- Owners: Configurable array of addresses
- Required confirmations: Configurable number
- Transaction time delay: Configurable duration

## 3. Methodology

The audit was conducted using the following approach:

- **Manual Code Review**: Line-by-line analysis to identify logical errors, security vulnerabilities, and gas inefficiencies.
- **Scenario Analysis**: Evaluating edge cases, such as malicious inputs, reentrancy attacks, and unexpected behaviors.
- **Best Practices Check**: Comparing the contract against industry standards (e.g., OpenZeppelin guidelines, ConsenSys best practices).
- **Gas Optimization Analysis**: Identifying opportunities to reduce gas consumption.

No automated tools were explicitly used, but findings were validated through logical reasoning and hypothetical attack scenarios.

# 4. Findings

## 4.1 High Severity

### MS-001: Reentrancy Vulnerability in `executeTransaction`

- **Severity**: High
- **Status**: resolved
- **Description**: The `executeTransaction` function is vulnerable to a reentrancy attack due to the incorrect order of state updates and external calls. The external call to `transactions[txId].to.call` is made before setting `transactions[txId].executed = true`. A malicious contract at the destination address ( `to` ) could re-enter `executeTransaction` multiple times for the same `txId`, potentially draining the contract's entire balance before the `executed` flag is updated.
- **Impact**: Complete loss of contract funds due to repeated execution of a single transaction.
- **Proof of Concept**:
  1. A malicious contract is proposed as the destination ( `to` ) in a

transaction.
2. The transaction is confirmed by the required number of owners.
3. During `executeTransaction`, the malicious contract's `fallback` or `receive` function re-calls `executeTransaction` before `executed` is set to `true`.
4. The transaction is executed multiple times, transferring the contract’s balance to the attacker.

- **Recommendation**:
  - Follow the Checks-Effects-Interactions pattern by setting `transactions[txId].executed = true` before the external call.
  - Alternatively, implement a `nonReentrant` modifier to prevent reentrancy.
  - Example fix:

```
function executeTransaction(uint256 txId) public {
    require(transactions[txId].to != address(0), "Wrong
    require(!transactions[txId].executed, "Transaction 
    require(confirmationCount[txId] >= required, "Insuf
    require(block.timestamp >= timeStarted[txId], "Time

    transactions[txId].executed = true; // Set before e

    (bool success, ) = transactions[txId].to.call{
        value: transactions[txId].value
    }(transactions[txId].data);
    require(success, "Transaction execution failed");

    emit txExecuted(txId, msg.sender);
}
```

## 4.2 Medium Severity

### MS-002: Bypassing Transaction Time Delay via `revokeTransaction`

- **Severity**: Medium to High

- **Status**: resolved
- **Description**: In a multi-signature wallet with `required = 2` and three owners, if all three owners confirm a transaction, the `timeStarted[txId]` is set to `block.timestamp + transactionTime`. If one owner calls `revokeTransaction`, it reduces `confirmationCount[txId]` by 1 (e.g., from 3 to 2) and sets `timeStarted[txId] = 0`. Since `confirmationCount[txId]` is still sufficient (2 >= required), the transaction remains eligible for execution. However, because `timeStarted[txId] = 0`, the check `block.timestamp >= timeStarted[txId]` always passes, allowing immediate execution and bypassing the intended time delay.
- **Impact**: Undermines the time delay mechanism, allowing premature transaction execution, which could violate the contract’s security model and trust assumptions among owners.
- **Proof of Concept**:
  1. A transaction is proposed and confirmed by all three owners, setting `timeStarted[txId] = block.timestamp + transactionTime`.
  2. One owner calls `revokeTransaction`, setting `timeStarted[txId] = 0` while `confirmationCount[txId] = 2`.
  3. The transaction can now be executed immediately via `executeTransaction`, bypassing the delay.
- **Recommendation**:
  - Remove the line `timeStarted[txId] = 0` from the `revokeTransaction` function as it is unnecessary.

## 4.3 Low Severity

### OPT-001: Gas Optimization by Using Mapping Instead of Array for Transactions

- **Severity**: Low
- **Status**: resolved

- **Description**: The contract stores transactions in a dynamic array ( `Transaction[] transactions` ), using `push` to add new transactions. This approach incurs higher gas costs due to dynamic array resizing, especially for large numbers of transactions. Using a `mapping(uint256 => Transaction)` with a counter ( `transactionCounter` ) would avoid array resizing costs and simplify transaction management.
- **Impact**: Increased gas costs for adding transactions, particularly in high-usage scenarios. Management of old transactions (e.g., cleanup) is also more complex with arrays.
- **Recommendation**:
  - Replace the `transactions` array with a mapping and use a `transactionCounter` to generate `txId` .
  - Example fix:

```
mapping(uint256 => Transaction) public transactions;
uint256 public transactionCounter;

function proposeTransaction(address _to, uint256 _value
    external ONLY_OWNER returns (uint256) {
    uint256 txId = transactionCounter;
    transactions[txId] = Transaction({
        to: _to,
        value: _value,
        data: _data,
        executed: false
    });
    transactionCounter++;
    emit txProposed(txId, msg.sender, _to, _value, _dat
    return txId;
}
```

## 4.4 Informational

### INF-001: Missing Zero Address Check for `_owners` in Constructor

- **Severity**: Informational

- **Status**: resolved
- **Description**: The `constructor` checks that `msg.sender` is not the zero address but does not verify that addresses in the `_owners` array are non-zero. Including a zero address as an owner could lead to unexpected behavior, such as invalid confirmations.
- **Impact**: Low risk, as developers typically provide valid addresses, but adding the check improves robustness.
- **Recommendation**:
    - Add a check in the `constructor` to ensure no zero addresses in `_owners`.
    - Example fix:

    ```
    for (uint256 i = 0; i < owners.length; i++) {
        require(owners[i] != address(0), "Owner cannot be z
        isOwner[owners[i]] = true;
    }
    ```

#### INF-002: Redundant Check for Negative `transactionTime`

- **Severity**: Informational
- **Status**: resolved
- **Description**: The `constructor` includes a check `require(_transactionTime >= 0, "Delay cannot be negative")`. Since `_transactionTime` is of type `uint256`, it cannot be negative, making this check redundant.
- **Impact**: Minor gas inefficiency and reduced code clarity.
- **Recommendation**:
    - Remove the redundant check:

    ```
    // Remove: require(_transactionTime >= 0, "Delay canno
    ```

# 5. Recommendations

1. **Address High and Medium Severity Issues**:
    - Implement the reentrancy fix for `executeTransaction` (MS-001) by following the Checks-Effects-Interactions pattern or using a `nonReentrant` modifier.
    - Modify `revokeTransaction` to prevent bypassing the time delay (MS-002) by only resetting `timeStarted` when confirmations fall below `required`.
2. **Apply Gas Optimization**:
    - Replace the `transactions` array with a mapping and use a `transactionCounter` (OPT-001) to reduce gas costs for transaction storage.
3. **Enhance Robustness**:
    - Add zero address checks for `_owners` in the `constructor` (INF-001).
    - Remove the redundant `transactionTime` check (INF-002) for better code clarity and minor gas savings.
4. **Testing and Validation**:
    - Test the contract with tools like Foundry or Hardhat to verify fixes for MS-001 and MS-002.
    - Simulate edge cases, such as malicious contracts, zero addresses, and high transaction volumes.
    - Use gas profiling tools to quantify the impact of OPT-001.
5. **Documentation**:
    - Update the contract documentation to clarify the time delay mechanism, owner management, and transaction lifecycle.
    - Use consistent and professional error messages to improve debugging.

# 6. Conclusion

The `MultiSig.sol` contract provides a functional multi-signature wallet but contains critical vulnerabilities that could lead to fund loss (MS-001) or bypassing of security mechanisms (MS-002). Additionally, one gas optimization (OPT-001) and two informational issues (INF-001, INF-002) were identified to enhance efficiency and robustness. Implementing the recommended fixes will significantly improve the contract's security and usability. The auditor recommends immediate action on high and medium-severity issues and consideration of informational improvements based on project requirements.

# 7. About the Auditor

MahdiFa is a smart contract auditor with experience in auditing Solidity-based contracts. Specializing in identifying security vulnerabilities, gas optimizations, and best practices, the auditor has conducted reviews for various blockchain projects, ensuring robust and secure smart contract deployments.

**Contact**: karizmeh811@proton.me

---

**Disclaimer**: This audit does not guarantee the absence of all vulnerabilities. It is recommended to conduct additional testing and formal verification before deploying the contract to production.