MultiSig Timelock Audit Report

Version 1.0

Tanu Gupta

December 22, 2025

MultiSig Timelock Audit Report, Codehawks

Tanu Gupta

December 22 , 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

- Medium
  - [M-1] Revoked signers' confirmations persist leading to unauthorized transaction execution
  - [M-2] Owner can DOS by limiting signers to less than or equal to required confirmations
- Low
  - [L-1] Self-revocation of signing role by owner can lead to loss of control
  - [L-2] Missing emergency/cancel function ability for pending transactions
  - [L-3] OpenZeppelin Version Mismatch Creates Silent Compiler Differences

Protocol Summary

This project is a secure, role-based multi-signature wallet with a built-in dynamic timelock mechanism, designed to add an extra layer of protection and governance for Ethereum funds, especially when dealing with large transaction amounts.

The wallet requires multiple authorized signers to approve transactions before execution, ensuring that no single entity can unilaterally control the funds. The dynamic timelock feature adjusts the waiting period for transaction execution based on the transaction amount, with larger amounts incurring longer delays.

This mechanism provides an additional safeguard against impulsive or unauthorized transactions, allowing time for review and potential cancellation if necessary. Overall, this multi-signature wallet with dynamic timelock functionality aims to enhance security and trust in managing Ethereum assets.

Disclaimer

I, Tanu make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Used the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The audit was performed between December 20, 2025 and December 20, 2025.

Scope

All Contracts in `src` are in scope.

- src/MultiSigTimelock.sol

Roles

Contract Owner (Deployer)

- The account that deploys the `MultiSigTimelock` contract.
- Automatically receives both the OpenZeppelin `DEFAULT_ADMIN_ROLE` and the custom `SIGNING_ROLE`, becoming the first signer.
- **Powers**:
  - Propose new transactions (recipient, value, data)
  - Grant the `SIGNING_ROLE` to additional addresses (up to a maximum of 5 total signers)
  - Revoke the `SIGNING_ROLE` from any signer except when it would drop the total below 1 (prevents bricking the wallet)
  - As a signer, can confirm transactions, revoke own confirmations, and execute transactions once quorum and timelock are satisfied

- **Limitations**:
  - Cannot unilaterally execute transactions — still requires 2 additional confirmations (minimum 3-of-N)
  - Cannot remove the last remaining signer
  - Cannot bypass the timelock delays for large transactions

Signers (holders of SIGNING_ROLE)

- Up to 5 addresses in total (owner + up to 4 others) that possess the `SIGNING_ROLE`.
- **Powers**:
  - Confirm pending transaction proposals
  - Revoke their own previous confirmation (useful if they change their mind before execution)
  - Execute a transaction once:
    * At least 3 distinct signers have confirmed, and
    * The value-based timelock period has fully elapsed
  - Propose new transactions (permission is tied to the role, so any signer can propose)
- **Limitations**:
  - Cannot grant or revoke roles — only the owner (admin) can manage membership
  - Cannot execute a transaction without meeting the 3-confirmation quorum and timelock requirement
  - No individual signer has more power than any other once the role is granted

Executive Summary

This report corresonds to the code available at the git repository at 2025-12-multisig-timelock.

Issues found

Severity	Number of issues found
High	2

Severity	Number of issues found
Medium	2
Low	3
Info	0
Total	7

Findings

High

[H-1] Owner-only proposal violates multi-sig specification leading to centralization risk

Description The specification states: *Propose new transactions (permission is tied to the role, so any signer can propose)*. However, the `MultiSigTimelock::proposeTransaction` function has an `onlyOwner` modifier, making it owner-exclusive.

```
1  function proposeTransaction(address to, uint256 value, bytes calldata
       data)
2          external
3          nonReentrant
4          noneZeroAddress(to)
5  @>      onlyOwner
6          returns (uint256)
7      {
8          return _proposeTransaction(to, value, data);
9      }
```

Impact The imapct is critical. This transforms the contract from a *true multi-signature* wallet to an *owner-controlled* wallet with advisory confirmations. The owner becomes a single point of failure/control, defeating the core purpose of multi-signature security.

Proof of Concepts

- The owner can unilaterally propose any transaction, and then simply wait for 2 other signers (totalling to 3) to confirm it, effectively controlling all outgoing transactions.
- This centralization risk means that if the owner's key is compromised, an attacker can propose malicious transactions and have them confirmed by colluding signers, leading to potential fund loss.

In this poc, as demonstrated in the test case below, only the owner can propose transactions. If a non-owner attempts to propose a transaction, it results in a revert, showcasing the violation of the multi-signature principle.

Paste the following test case into `test/MultiSigTimelock.t.sol` to reproduce:

```
function testOnlyOwnerCanProposeTransaction() public {
        vm.deal(OWNER, OWNER_BALANCE_ONE);

        vm.prank(OWNER);
        uint256 txnId = multiSigTimelock.proposeTransaction(SPENDER_ONE
            , OWNER_BALANCE_ONE, hex"");
        assertEq(txnId, 0);

        vm.prank(OWNER);
        uint256 txnIdTwo = multiSigTimelock.proposeTransaction(
            SPENDER_ONE, OWNER_BALANCE_ONE, hex"");
        assertEq(txnIdTwo, 1);

        address nonOwner = makeAddr("non_owner");
        vm.prank(nonOwner);
        //Expect revert since only owner can propose making multi-sig
            ineffective
        vm.expectRevert();
        multiSigTimelock.proposeTransaction(SPENDER_ONE,
            OWNER_BALANCE_ONE, hex"");
    }
```

Recommended mitigation Replace `onlyOwner` with `onlyRole(SIGNING_ROLE)` in `MultiSigTimelock::proposeTransaction`

```
function proposeTransaction(address to, uint256 value, bytes calldata
    data)
        external
        nonReentrant
        noneZeroAddress(to)
-       onlyOwner
+       onlyRole(SIGNING_ROLE)
        returns (uint256)
    {
        return _proposeTransaction(to, value, data);
    }
```

[H-2] Signers can delay confirmations indefinitely to bypass timelock then execute proposals immediately, leading to governance attack

Description Signers can strategically withhold confirmations until **AFTER** the timelock period expires, then confirm and execute immediately. This completely defeats the purpose of the timelock as a

cooling-off period.

The timelock clock should start after sufficient confirmations, not after proposal. Currently:

- Timelock starts: When transaction is proposed
- Timelock should start: When quorum is reached

Impact HIGH. The timelock becomes completely ineffective against coordinated signers. Malicious actors can:

- Propose a transaction (with the understanding that signers are allowed to propose as per spec)
- Wait for timelock to expire
- Confirm and execute immediately

The intended design of **waiting period for reconsideration** is completely eliminated.

Proof of Concepts

- Deploy the contract with multiple signers.
- Propose a transaction and do not confirm it.
- Advance time beyond the timelock period.
- Have the required signers confirm the transaction.
- Execute the transaction immediately after confirmations.
- The transaction executes without any delay, demonstrating that the timelock was effectively bypassed.

Paste the following test case into `test/MultiSigTimelock.t.sol` to reproduce:

```
1  function testBypassTimelockByDelayingConfirmationsTillExpiry()
       external {
2          vm.startPrank(OWNER);
3          multiSigTimelock.grantSigningRole(SIGNER_TWO);
4          multiSigTimelock.grantSigningRole(SIGNER_THREE);
5          multiSigTimelock.grantSigningRole(SIGNER_FOUR);
6          multiSigTimelock.grantSigningRole(SIGNER_FIVE);
7          vm.stopPrank();
8
9          uint256 amountToSend = 100 ether;
10         vm.deal(address(multiSigTimelock), amountToSend);
11         vm.deal(OWNER, amountToSend);
12         vm.prank(OWNER);
13         uint256 txnId = multiSigTimelock.proposeTransaction(SPENDER_ONE
               , amountToSend, hex"");
14
15         vm.expectRevert(); //
               MultiSigTimelock__InsufficientConfirmations(3,0)
16         multiSigTimelock.executeTransaction(txnId);
17
```

```
        uint256 timelockDelay = 7 days;
        vm.warp(block.timestamp + timelockDelay);

        vm.prank(OWNER);
        multiSigTimelock.confirmTransaction(txnId);
        vm.prank(SIGNER_TWO);
        multiSigTimelock.confirmTransaction(txnId);
        vm.prank(SIGNER_THREE);
        multiSigTimelock.confirmTransaction(txnId);

        multiSigTimelock.executeTransaction(txnId);
    }
```

Recommended mitigation

- Track *Confirmed At* timestamp for each transaction by adding this field to the `Transaction` struct.
- Start the timelock countdown only after the transaction reaches the required confirmations.
- Modify `executeTransaction` to check if the current time is greater than `Confirmed At + TIMELOCK_DURATION`.
- This ensures that even if signers delay their confirmations, the timelock period is always enforced after quorum is achieved.

```
struct Transaction {
    address to;
    uint256 value;
    bytes data;
    uint256 confirmations;
    uint256 proposedAt;
+   uint256 confirmedAt; // NEW: When quorum was reached
    bool executed;
}

function _confirmTransaction(uint256 txnId) internal {

        if (s_signatures[txnId][msg.sender]) {
            revert MultiSigTimeLock__UserAlreadySigned();
        }
        s_signatures[txnId][msg.sender] = true;
        s_transactions[txnId].confirmations++;

+       if (s_transactions[txnId].confirmations ==
    REQUIRED_CONFIRMATIONS) {
+       s_transactions[txnId].confirmedAt = block.timestamp;
    }

        emit TransactionConfirmed(txnId, msg.sender);
    }

```

```
26 function _executeTransaction(uint256 txnId) internal {
27     Transaction storage txn = s_transactions[txnId];
28
29     if (txn.confirmations < REQUIRED_CONFIRMATIONS) {
30         revert MultiSigTimelock__InsufficientConfirmations(
               REQUIRED_CONFIRMATIONS, txn.confirmations);
31     }
32
33 +    // NEW: Timelock starts from confirmation, not proposal
34     uint256 requiredDelay = _getTimelockDelay(txn.value);
35 -   uint256 executionTime = txn.proposedAt + requiredDelay;
36 +   uint256 executionTime = txn.confirmedAt + requiredDelay; // NOT txn
       .proposedAt
37     if (block.timestamp < executionTime) {
38         revert MultiSigTimelock__TimelockHasNotExpired(executionTime);
39     }
40
41 +    // ... rest of execution logic
42 }
```

Medium

[M-1] Revoked signers' confirmations persist leading to unauthorized transaction execution

Description When a signer is revoked via `MultiSigTimelock::revokeSigningRole`, their existing confirmations on pending transactions are not removed. These orphaned confirmations continue to count toward the `REQUIRED_CONFIRMATIONS` threshold.

```
1  function revokeSigningRole(address _account) external nonReentrant
       onlyOwner noneZeroAddress(_account) {
2          // CHECKS
3          if (!s_isSigner[_account]) {
4              revert MultiSigTimelock__AccountIsNotASigner();
5          }
6          // Prevent revoking the first signer (would break the multisig)
               , moreover, the first signer is the owner of the contract(
               wallet)
7          if (s_signerCount <= 1) {
8              revert MultiSigTimelock__CannotRevokeLastSigner();
9          }
10
11         // Find the index of the account in the array
12         uint256 indexToRemove = type(uint256).max; // Use max as "not
               found" indicator
13         for (uint256 i = 0; i < s_signerCount; i++) {
14             if (s_signers[i] == _account) {
15                 indexToRemove = i;
```

```
16                  break;
17              }
18          }
19
20          // Gas-efficient array removal: move last element to removed
                position
21          if (indexToRemove < s_signerCount - 1) {
22              // Move the last signer to the position of the removed
                    signer
23              s_signers[indexToRemove] = s_signers[s_signerCount - 1];
24          }
25
26          // Clear the last position and decrement count
27          s_signers[s_signerCount - 1] = address(0);
28          s_signerCount -= 1;
29
30          s_isSigner[_account] = false;
31      @>      _revokeRole(SIGNING_ROLE, _account);
32      }
```

Impact HIGH. Security dilution over time. A malicious signer can confirm many transactions, get revoked, and their confirmations still reduce the remaining requirement. This can lead to unauthorized execution of transactions with fewer active signers than intended.

Example: With 5 signers (REQUIRED=3), if a revoked signer confirmed a transaction, it now only needs 2 more confirmations from 4 active signers instead of 3.

Proof of Concepts

1. Deploy the contract with 5 signers.
2. Propose a transaction and have 3 signers confirm it.
3. Revoke the signing role of two of the confirming signers.
4. Execute the transaction successfully with only 1 remaining active signer confirming it, due to the orphaned confirmations from the revoked signers.

Paste the following test case into `test/MultiSigTimelock.t.sol` to reproduce:

```
1   function testRevokedSignerConfirmationsRemainsForPendingTransactions
        () external grantSigningRoles {
2           // ARRANGE
3           vm.deal(address(multiSigTimelock), OWNER_BALANCE_ONE);
4           vm.deal(OWNER, OWNER_BALANCE_ONE);
5           vm.prank(OWNER);
6           uint256 txnId = multiSigTimelock.proposeTransaction(SPENDER_ONE
                , OWNER_BALANCE_ONE, hex"");
7
8           vm.prank(OWNER);
9           multiSigTimelock.confirmTransaction(txnId);
10          vm.prank(SIGNER_TWO);
```

```
        multiSigTimelock.confirmTransaction(txnId);
        vm.prank(SIGNER_THREE);
        multiSigTimelock.confirmTransaction(txnId);

        //Due to the malicious behaviour of SIGNER_TWO and SIGNER_THREE
            , owner has revoked their signing roles
        multiSigTimelock.revokeSigningRole(SIGNER_TWO);
        multiSigTimelock.revokeSigningRole(SIGNER_THREE);

        vm.prank(OWNER);
        //confirmations of revoked signers are being considered here as
            well
        multiSigTimelock.executeTransaction(txnId);

    }
```

Recommended mitigation In `MultiSigTimelock::revokeSigningRole`, iterate through pending transactions and remove the revoked signer's confirmations:

```
for (uint256 i = 0; i < s_transactionCount; i++) {
    if (!s_transactions[i].executed && s_signatures[i][_account]) {
        s_signatures[i][_account] = false;
        s_transactions[i].confirmations--;
    }
}
```

[M-2] Owner can DOS by limiting signers to less than or equal to required confirmations

Description The owner controls adding signers but `REQUIRED_CONFIRMATIONS` is fixed at 3. The owner can add only 2 additional signers (total 3), making transactions require 3/3 confirmations, allowing any single dissenting signer to block all transactions.

```
function grantSigningRole(address _account) external nonReentrant
    onlyOwner noneZeroAddress(_account) {
        if (s_isSigner[_account]) {
            revert MultiSigTimelock__AccountIsAlreadyASigner();
        }

        if (s_signerCount >= MAX_SIGNER_COUNT) {
            revert MultiSigTimelock__MaximumSignersReached();
        }

        s_signers[s_signerCount] = _account;
        s_isSigner[_account] = true;
        s_signerCount += 1;

        _grantRole(SIGNING_ROLE, _account);
    }
```

Impact HIGH. Complete governance paralysis. The owner (or a malicious actor if ownership is compromised) can permanently disable the multi-sig functionality.

Proof of Concepts

1. Deploy the contract with only the owner as signer.
2. Owner adds 2 more signers (total 3).
3. With only 2 confirmations, no transaction can reach the required 3 confirmations, effectively locking all indefinitely.

Paste the following test case into `test/MultiSigTimelock.t.sol` to reproduce:

```
        function testLimitedSignersCanCauseDOS() external {
         vm.startPrank(OWNER);
         multiSigTimelock.grantSigningRole(SIGNER_TWO);
         multiSigTimelock.grantSigningRole(SIGNER_THREE);
         vm.stopPrank();

         // ARRANGE
         vm.deal(address(multiSigTimelock), OWNER_BALANCE_ONE);
         vm.deal(OWNER, OWNER_BALANCE_ONE);
         vm.prank(OWNER);
         uint256 txnId = multiSigTimelock.proposeTransaction(SPENDER_ONE
             , OWNER_BALANCE_ONE, hex"");


         vm.prank(OWNER);
         multiSigTimelock.confirmTransaction(txnId);
         vm.prank(SIGNER_TWO);
         multiSigTimelock.confirmTransaction(txnId);

         vm.prank(SIGNER_TWO);
         vm.expectRevert();
         //confirmations of revoked signers are being considered here as
             well
         multiSigTimelock.executeTransaction(txnId);
     }
```

Recommended mitigation

- Make signer addition/removal require multi-sig confirmation
- Signers should be able to propose adding/removing signers
- Siners should be penalized for inactivity to prevent long-term DOS

Low

[L-1] Self-revocation of signing role by owner can lead to loss of control

Description The owner can revoke their own signing role via `MultiSigTimelock::revokeSigningRole`. While this doesn't remove ownership, it prevents them from confirming/revoking transactions or executing proposals.

Though it is possible to regain signing role via `MultiSigTimelock::grantSigningRole`, as only the owner can do this. If the owner loses access to their key, they cannot recover signing capabilities.

Impact While ownership remains, the owner loses signing capabilities. If this happens accidentally, they cannot participate in the multi-sig process without granting themselves the role again (which they can do as owner). That's why the likelyhood is low.

Proof of Concepts

- Owner grants signing roles to multiple signers.
- Owner proposes a transaction.
- Owner revokes their own signing role.
- Owner can no longer confirm or execute the proposed transaction.
- Can only regain signing role by calling `grantSigningRole` on themselves. If the owner's key is lost, they cannot recover signing capabilities.

Paste the following test case into `test/MultiSigTimelock.t.sol` to reproduce:

```
1   function test_if_Owner_can_revoke_his_signing_role() external {
2          vm.startPrank(OWNER);
3         multiSigTimelock.grantSigningRole(SIGNER_TWO);
4         multiSigTimelock.grantSigningRole(SIGNER_THREE);
5         multiSigTimelock.grantSigningRole(SIGNER_FOUR);
6         multiSigTimelock.grantSigningRole(SIGNER_FIVE);
7         vm.stopPrank();
8
9         vm.deal(address(multiSigTimelock), OWNER_BALANCE_ONE);
10        vm.deal(OWNER, OWNER_BALANCE_ONE);
11        vm.prank(OWNER);
12        multiSigTimelock.proposeTransaction(SPENDER_ONE,
              OWNER_BALANCE_ONE, hex"");
13
14        vm.startPrank(OWNER);
15        multiSigTimelock.revokeSigningRole(OWNER);
16        vm.stopPrank();
17    }
```

Recommended mitigation Add explicit check to prevent owner from revoking their own signing role from `MultiSigTimelock::revokeSigningRole`

```
function revokeSigningRole(address _account) external nonReentrant
    onlyOwner noneZeroAddress(_account) {
        // CHECKS
        if (!s_isSigner[_account]) {
            revert MultiSigTimelock__AccountIsNotASigner();
        }
        // Prevent revoking the first signer (would break the multisig)
            , moreover, the first signer is the owner of the contract(
            wallet)
        if (s_signerCount <= 1) {
            revert MultiSigTimelock__CannotRevokeLastSigner();
        }
+       if (_account == owner()) {
+           revert MultiSigTimelock__CannotRevokeOwner();
+       }

        [...]
        _revokeRole(SIGNING_ROLE, _account);
    }
```

[L-2] Missing emergency/cancel function ability for pending transactions

Description No mechanism exists to cancel/update a proposed transaction, even if it contains errors or becomes obsolete. Transactions must either execute (potentially incorrectly) or remain stuck forever.

Impact

- Poor user experience
- Potential fund loss if transactions contain errors. Could lead to permanently stuck proposals cluttering the system.

Recommended mitigation Add `cancelTransaction(uint256 txnId)` function callable by owner or via multi-sig to remove pending transactions.

[L-3] OpenZeppelin Version Mismatch Creates Silent Compiler Differences

Description The `MultiSigTimelock` contract uses `pragma solidity ^0.8.19;` while importing `OpenZeppelin` contracts that use `pragma solidity ^0.8.20;`. This creates a compiler version mismatch where the contract compiles with Solidity `0.8.19`, but the imported OpenZeppelin libraries are designed for `0.8.20`.

1 Found Instances

- Found in src/MultiSigTimelock.sol Line: 27

```
1  pragma solidity ^0.8.19;
```

Impact This mismatch can lead to silent differences in compiler behavior, optimizations, and security features between the contract and the imported libraries. Potential risks include:

- Inconsistent handling of certain language features or optimizations
- Unexpected vulnerabilities due to differing compiler assumptions
- Difficulty in auditing and verifying the contract due to version discrepancies

Recommended mitigation Align the Solidity pragma version in `MultiSigTimelock.sol` with that of the imported OpenZeppelin contracts by updating it to `pragma solidity ^0.8.20;`.

```
1 - pragma solidity ^0.8.19;
2 + pragma solidity ^0.8.20;
```