Kode-n-Rolla

MultiSig Timelock Security Review

Dec 18th 2025 - Dec 25th 2025

Table of Contents

Whoami

I am Pavel (aka *kode-n-rolla*), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Solidity*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Risk Classification

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

I used *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
  - Funds are directly or nearly directly at risk
  - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
  - Funds are indirectly at risk.

  - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
  - Funds are not at risk.
  - However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

The project implements a **role-based multi-signature wallet** with a **dynamic, value-dependent timelock** designed to secure Ethereum funds and reduce the risk of rushed or compromised high-value transactions.

The wallet follows a standard multi-signature execution model, requiring a minimum quorum of signers (3 out of up to 5) to approve transactions before execution. On top of this, it introduces an **automatic timelock mechanism** where the delay before execution increases proportionally with the ETH value of the transaction. This design aims to balance usability for low-value transfers with stronger safety guarantees for high-value operations.

Access control and signer permissions are enforced using **OpenZeppelin's *AccessControl***, enabling granular role separation between transaction proposers, confirmers, and administrators.

The protocol exposes a transparent lifecycle for transactions — proposal, confirmation, delay, and execution — supported by on-chain events and public getters, allowing off-chain systems and users to easily track wallet state and pending actions.

Key Characteristics

- Up to **5 designated signers**, with a **3-of-5 minimum confirmation threshold**
- **Role-based access control** for governance and transaction management
- **Dynamic timelock** based on transaction value:
  - *< 1 ETH* → no delay
  - *1–10 ETH* → 1-day delay
  - *10–100 ETH* → 2-day delay
  - *≥ 100 ETH* → 7-day delay
- Event-driven proposal → confirmation → execution workflow
- Public view functions for transparent state inspection

Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the *CodeHawks* platform. The assessment focused exclusively on the ***Solidity*** smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "MultiSig Timelock" *CodeHawks* audit event.

Scope

The audit scope consisted of the following smart contract:

```
src/
├── MultiSigTimelock.sol
```

Roles

- Contract Owner (Deployer)

The account that deploys the *MultiSigTimelock* contract.

Upon deployment, the owner is assigned both the OpenZeppelin *DEFAULT_ADMIN_ROLE* and the custom *SIGNING_ROLE*, making it the initial signer.

Permissions

- Propose new transactions (recipient, ETH value, calldata).
- Grant the *SIGNING_ROLE* to additional addresses, up to a maximum of **5 total signers**.
- Revoke the *SIGNING_ROLE* from existing signers, except when such revocation would reduce the total number of signers below one.
- As a signer, confirm transactions, revoke its own confirmations, and execute transactions once quorum and timelock conditions are met.

Limitations

- Cannot unilaterally execute transactions; a minimum of **3 distinct signer confirmations** is always required.
- Cannot remove the last remaining signer.
- Cannot bypass or shorten timelock delays for high-value transactions.

- Signers (*SIGNING_ROLE* holders)

Addresses holding the *SIGNING_ROLE*. The total number of signers is capped at **5**, including the contract owner.

Permissions

- Propose new transactions.
- Confirm pending transaction proposals.
- Revoke their own confirmations prior to execution.
- Execute a transaction once:
  - At least **3 distinct signers** have confirmed the transaction, and
  - The value-dependent timelock period has fully elapsed.

Limitations

- Cannot grant or revoke roles; role management is restricted to the admin (DEFAULT_ADMIN_ROLE).
- Cannot execute transactions without satisfying both the quorum and timelock requirements.
- No individual signer has elevated privileges beyond those granted by the role.

Executive Summary

The audit identified **two Medium-severity issues** affecting the governance and safety guarantees of the multi-signature wallet during **signer rotation.**

Both findings relate to how transaction confirmations are handled when the signer set changes. Specifically, the contract does not fully enforce the intended **"3-of-N active signers"** security model once signers are revoked, and it lacks safe mechanisms to clean up or invalidate stale approvals.

The first issue allows transactions to remain executable based on confirmations from **revoked signers**, potentially enabling execution even when the number of active signers falls below the documented minimum threshold. The second issue makes such stale confirmations **irrevocable**, as revoked signers lose the ability to withdraw confirmations they submitted while authorized.

While neither issue enables immediate fund theft in isolation, together they **weaken the multisig's governance model**, undermine signer rotation as a security control, and increase the risk of unintended or unsafe transaction execution in real-world operational scenarios.

Issue found

Severity	Issue
Medium	Signer removal does not invalidate prior confirmations, allowing execution with fewer than 3 active signers
Medium	Signer revocation makes prior confirmations irrevocable, amplifying stale-approval risks

Findings

Mediums

[M-01] Signer removal does not invalidate prior confirmations, allowing execution with fewer than 3 active signers

Description

Confirmations are tracked using a cumulative counter (*Transaction.confirmations*) that is incremented in *_confirmTransaction()*. However, when a signer is removed via *revokeSigningRole()*, existing confirmations are **not invalidated**, and the stored counter is **not adjusted**.

During execution, *_executeTransaction()* checks only that:

```
txn.confirmations >= REQUIRED_CONFIRMATIONS
```

without verifying whether those confirmations correspond to **currently active signers**.

As a result, a transaction can remain executable even after one or more confirming signers have been revoked, including scenarios where the total number of active signers drops below the intended minimum of 3.

```
    function revokeSigningRole(address _account) external nonReentrant onlyOwner
noneZeroAddress(_account) {
...
@>      // No logic for:
        //    s_signatures[txnId][_account] = false
        //    decrease s_transactions[txnId].confirmations

        // Clear the last position and decrement count
        s_signers[s_signerCount - 1] = address(0);
        s_signerCount -= 1;

        s_isSigner[_account] = false;
        _revokeRole(SIGNING_ROLE, _account);
    }
    function _executeTransaction(uint256 txnId) internal {
        Transaction storage txn = s_transactions[txnId];

@>          // No check for 3 current signers signed
        // CHECKS
        // 1. Check if enough confirmations
        if (txn.confirmations < REQUIRED_CONFIRMATIONS) {
            revert
MultiSigTimelock__InsufficientConfirmations(REQUIRED_CONFIRMATIONS,
txn.confirmations);
        }
...
    }
```

Impact

Transactions may be executed based on confirmations from accounts that are **no longer authorized signers**, violating the intended **"3-of-N active signers"** security model.

This can place the multisig in a **weakened governance state**, where transactions remain executable even when the number of active signers falls below the documented minimum threshold, undermining signer rotation as an effective security control.

Likelihood

Signer set changes (granting and revoking the *SIGNING_ROLE*) are part of the **normal multisig lifecycle** and are explicitly supported by the contract design.

Because confirmations are tracked cumulatively and never recalculated when signers are revoked, the issue manifests naturally during standard usage and **does not require precise timing, malicious coordination, or edge-case behavior.**

Proof of Concept

```
    function test_TxExecutableWithFewerThanRequiredActiveSigners() public {
        uint256 value = 0.5 ether;

        // Fund wallet for the transfer (< 1 ETH => no timelock)
        vm.deal(address(multiSigTimelock), value);
        // Add 2 signers (3 total including owner)
        multiSigTimelock.grantSigningRole(SIGNER_TWO);
        multiSigTimelock.grantSigningRole(SIGNER_THREE);
        assertEq(multiSigTimelock.getSignerCount(), 3);

        // Propose tx
        vm.prank(OWNER);
        uint256 txnId = multiSigTimelock.proposeTransaction(SPENDER_ONE,
OWNER_BALANCE_ONE, hex"");

        // 3 confirmations
        vm.prank(OWNER);
        multiSigTimelock.confirmTransaction(txnId);
        vm.prank(SIGNER_TWO);
        multiSigTimelock.confirmTransaction(txnId);
        vm.prank(SIGNER_THREE);
        multiSigTimelock.confirmTransaction(txnId);

        // Revoke signing role
        vm.prank(OWNER);
        multiSigTimelock.revokeSigningRole(SIGNER_TWO);

        // Only 2 active signers remain, but confirmations still count as 3
        MultiSigTimelock.Transaction memory t = multiSigTimelock.getTransaction(txnId);
        assertLt(multiSigTimelock.getSignerCount(),
multiSigTimelock.getRequiredConfirmations());
        assertEq(t.confirmations, 3);

        // Execute succeeds
        vm.prank(SIGNER_THREE);
        multiSigTimelock.executeTransaction(txnId);

        MultiSigTimelock.Transaction memory t2 = multiSigTimelock.getTransaction(txnId);
        assertTrue(t2.executed);
        assertEq(SPENDER_ONE.balance, value);
    }
```

Recommended Mitigation

Validate confirmations against the **current signer set at execution time**, rather than relying on a stored cumulative counter.

Instead of checking `Transaction.confirmations`, recompute the number of **valid confirmations** during execution:

- Iterate over the current signer list (`s_signers[0 .. s_signerCount - 1]`)

- Count entries where `s_signatures[txnId][signer] == true`
- Require `validCount >= REQUIRED_CONFIRMATIONS`

Since the signer set is capped at **5 addresses**, this loop has a constant and minimal cost, while ensuring that confirmations from revoked signers cannot satisfy the execution threshold.

```
    function _executeTransaction(uint256 txnId) internal {
        Transaction storage txn = s_transactions[txnId];

        // CHECKS
-       // 1. Check if enough confirmations
-       if (txn.confirmations < REQUIRED_CONFIRMATIONS) {
-           revert
MultiSigTimelock__InsufficientConfirmations(REQUIRED_CONFIRMATIONS,
txn.confirmations);
-       }
+        // 1. Check if enough *valid* confirmations from current signers
+        uint256 validConfirmations = _countValidConfirmations(txnId);
+        if (validConfirmations < REQUIRED_CONFIRMATIONS) {
+            revert
MultiSigTimelock__InsufficientConfirmations(REQUIRED_CONFIRMATIONS,
validConfirmations);
+        }

        // 2. Check if timelock period has passed
        uint256 requiredDelay = _getTimelockDelay(txn.value);
        uint256 executionTime = txn.proposedAt + requiredDelay;
        if (block.timestamp < executionTime) {
            revert MultiSigTimelock__TimelockHasNotExpired(executionTime);
        }
...
        emit TransactionExecuted(txnId, txn.to, txn.value);
    }

+   /// @dev Counts confirmations made by *current* signers only (bounded by
MAX_SIGNER_COUNT).
+   function _countValidConfirmations(uint256 txnId) internal view returns (uint256
count) {
+       for (uint256 i = 0; i < s_signerCount; i++) {
+           address signer = s_signers[i];
+           if (s_signatures[txnId][signer]) {
+               count++;
+           }
+       }
+   }
```

[M-02] Signer revocation makes prior confirmations irrevocable, amplifying stale-approval risks

Description

When a signer's role is revoked, they lose permission to call *revokeConfirmation*, even for confirmations they submitted while they were authorized. While this access control decision is individually reasonable, it causes prior confirmations to become **irreversible** after signer rotation, reducing operational safety and amplifying risks related to stale approvals.

Expected behavior

A signer who previously confirmed a transaction should be able to **revoke their own confirmation** as a safety measure, especially during signer rotation (e.g., key compromise, replacement, or inactivity), even if their signing role has since been revoked.

Actual behavior

The *revokeConfirmation* function is protected by *onlyRole(SIGNING_ROLE).* Once a signer's role is revoked, they permanently lose the ability to revoke any confirmations they previously submitted, even though those confirmations were created while the signer was fully authorized.

As a result:

- Prior confirmations become **stuck** after signer rotation.
- The multisig cannot reliably clean up or invalidate approvals from removed signers.
- This behavior increases reliance on implicit assumptions about signer stability over time.

While the access control restriction is locally consistent, it introduces broader **operational and governance risks** in dynamic signer environments.

```
function revokeConfirmation(uint256 txnId)
    external
    nonReentrant
    transactionExists(txnId)
    notExecuted(txnId)
@>  onlyRole(SIGNING_ROLE)
{
    _revokeConfirmation(txnId);
}
```

Impact

The primary impact is a loss of **governance and operational flexibility:** confirmations submitted by former signers cannot be withdrawn, even if circumstances change.

The direct security impact is limited on its own, but the issue **significantly amplifies the impact of stale-confirmation bugs,** particularly when combined with **[M-01]**, where confirmations from revoked signers can still count toward execution thresholds.

In combination, these behaviors weaken the intended multisig safety model during signer rotation.

Likelihood

Signer rotation (revoking and replacing signers) is a **standard operational practice** for multisig wallets, commonly performed in response to key compromise, signer inactivity, or organizational changes.

The issue manifests deterministically whenever:

1. A signer confirms a transaction, and
2. That signer is later revoked.

At that point, `revokeConfirmation` becomes permanently unavailable to the original signer due to the `onlyRole(SIGNING_ROLE)` restriction.

Proof of Concept

```
    function test_NoRevokeConfirmationAfterRevokeSignerRole() public {
        // Add 1 signer
        multiSigTimelock.grantSigningRole(SIGNER_TWO);

        // Propose tx
        vm.prank(OWNER);
        uint256 txnId = multiSigTimelock.proposeTransaction(SPENDER_ONE,
OWNER_BALANCE_ONE, hex"");

        // Confirm tx
        vm.prank(SIGNER_TWO);
        multiSigTimelock.confirmTransaction(txnId);

        // Revoke signing role
        vm.prank(OWNER);
        multiSigTimelock.revokeSigningRole(SIGNER_TWO);
        // Revoke confirmation
        vm.prank(SIGNER_TWO);
        vm.expectRevert();
        multiSigTimelock.revokeConfirmation(txnId);

        MultiSigTimelock.Transaction memory t = multiSigTimelock.getTransaction(txnId);
        assertEq(t.confirmations, 1);
    }
```

Recommended Mitigation

Allow an account to revoke **its own prior confirmation** even after its signing role has been revoked, while still preventing unauthorized revocation of confirmations by unrelated accounts.

Concretely:

- Replace the *onlyRole(SIGNING_ROLE)* access gate on *revokeConfirmation* with a custom check that allows:
    - Current signers to revoke confirmations (unchanged behavior), and
    - Former signers to revoke **only confirmations they personally submitted.**

This preserves *revokeConfirmation* as a safety mechanism during signer rotation, without granting revoked signers any broader execution or confirmation privileges.

```
    function revokeConfirmation(uint256 txnId)
        external
        nonReentrant
        transactionExists(txnId)
        notExecuted(txnId)
-       onlyRole(SIGNING_ROLE)
    {
+       // Allow either:
+       //  - current signers (SIGNING_ROLE), or
+       //  - former signers who previously confirmed this txn (can only revoke their own confirmation)
+       if (!hasRole(SIGNING_ROLE, msg.sender) && !s_signatures[txnId][msg.sender]) {
+           revert MultiSigTimelock__AccountIsNotASigner();
+       }
        _revokeConfirmation(txnId);
    }
```