Mute Smart Contract Final Audit Report

Scope of Audit

The scope of this audit was to analyze and document Mute smart contract codebase for quality, security, and correctness.

Check Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour
  .
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality and we recommend these issues to be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

TYPE	HIGH	MEDIUM	LOW	INFORMATIONAL
OPEN	0	0	0	0
CLOSED	3	2	2	0

Introduction

During the period of **March 25th, 2021 to March 30th, 2021** - Quillhash Team performed a security audit for **Mute** smart contracts.

The code for the audit was taken from following the official GitHub link**:**

https://github.com/muteio/mute-contracts/tree/main/contracts/Gov

https://github.com/muteio/mute-contracts/blob/main/contracts/Mute/MuteGovernance.sol

Commit Hash:

- **Governance Contracts - f9763b5c4ee61c903238463f87d39c98f6244587**
- **MuteGovernance.sol - e03c36c5e1db039be17876ea0b0ce2fda35ec539**

A. Contract Name - GovCoordinator & GovFunding

Issues Found – Code Review / Manual Testing

High Severity Issues

A.1 Contract completely locks Ether and fails to provide a way to Unlock it

Line no:

- **GovCoordinator.sol - 137**
- **GovFunding.sol - 147**

Status: CLOSED

Description:

The **execute function** of the contract includes the **payable** keyword. It indicates that the contract allows the transfer of **ETHER** into the contract.

However, no function to withdraw Ether from the contract was found.

This could lead to a very undesirable situation where any Ether sent to the contract will be completely lost and unrecoverable.

```
137     function execute(uint256 proposalId) public payable {
138         require(state(proposalId) == ProposalState.Succeeded, "GovCoordinator::execute:
139         Proposal storage proposal = proposals[proposalId];
140         proposal.executed = true;
141         (bool result, ) = address(proposal.target).call(proposal.data);
142         if (!result) {
143             revert("GovCoordinator::execute: transaction Failed");
144         }
145         emit ProposalExecuted(proposalId);
146     }
```

Recommendation:

The contract should either remove the **payable keyword** from the **execute function** or include a function that allows the withdrawal of the locked ETHER in the contract.

A.2 block.timestamp is being compared to block.number

Status: CLOSED

Line no:

- **GovCoordinator.sol - 170**
- **GovFunding.sol - 180**

Description:

In the **state function** of both the above-mentioned contract, **block.timestamp** is being compared to **proposal.endBlock** which is a **block.number.**

Is this Intended?

Since timestamp and block number represents different instances of a block, this if statement will fail to provide an accurate result.

```
170         } else if (block.timestamp >= proposal.endBlock) {
171             return ProposalState.Expired;
172         }
```

Recommendation:

block.number should be used instead of **block.timestamp.**

A.3 State of a Proposal ID will never reach ProposalState.Expired state

Line no:

- **GovCoordinator.sol - 170**
- **GovFunding.sol - 180**

Status: CLOSED

Description:

In the State function, the **state** of any particular proposal will never reach the **Expired state** even after **block.number** is greater than the **proposal.endBlock.**

```
178         } else if (proposal.executed == true) {
179             return ProposalState.Executed;
180         } else if (block.timestamp >= proposal.endBlock) {
181             return ProposalState.Expired;
182         }
```

This is because of the **else if** condition at line 178(*in the GovFunding.sol*) which returns **ProposalState.Executed** for every proposal that is executed.

Now, if a particular proposal is executed the ***else if*** condition will simply return **ProposalState.Executed** and stop further execution of the **State** function.

And, if a proposal is not executed, the ***else if*** condition at *Line 176* will return **ProposalState.Succeeded** and stop further execution of the function.

```
176         } else if (proposal.executed == false) {
177             return ProposalState.Succeeded;
```

Thus, never allowing the control flow of the State function to reach the **ProposalState.Expired state.**

Recommendation:

The **State** function must be rechecked and modified with adequate logic so that it shows the precise state of any Proposal ID.

Medium Severity Issues

A.4 Use require() instead of revert()

Status: CLOSED

Line no:

- **GovCoordinator.sol - 142**
- **GovFunding.sol - 152**

Description:

The **execute** function includes a **revert() statement** to ensure successful execution of the external call.

```
152         if (!result) {
153             revert("GovFunding::execute: transaction Failed");
154         }
```

However, it is considered a better practice to use require statements in order to ensure valid conditions, such as inputs or to validate return values from calls to external contracts.
While this enhances the readability of the code, it also effectively helps in gas optimization.

Recommendation:

The above-mentioned **if statement** can be modified to a **require** statement as follows:

require(result, ”GovFunding::execute: transaction Failed")

A.5 No Events emitted after imperative State Variable modification

Status: CLOSED

Line no:

- **GovCoordinator.sol - 87,93,99**
- **GovFunding.sol - 90,96,102**

Description:

Functions that update an imperative arithmetic state variable contract should emit an event after the updation.

The following functions modify some crucial arithmetic parameters like ***quorumVotes, votingPeriod, voteRequirement***

- ***changeQuorumVotes***
- ***changeProposalThreshold***
- ***changeVotingPeriod***

Since there is no event emitted on updating these variables, it might be difficult to track it off-chain.

***Recommendation*:**

An event should be fired after changing crucial arithmetic state variables.

Low Severity Issues

A.6 External visibility should be preferred

Status: Not Considered

Line no:

- **GovCoordinator.sol - 87,93,99,104,137,148,153,175,179**
- **GovFunding.sol - 90,96,102,107,147,158,163,185,189**

Description:

Those functions that are never called throughout the contract should be marked as ***external*** visibility instead of ***public*** visibility.

This will effectively result in Gas Optimization as well.

Therefore, the following function must be marked as **external** within the contract:

- ***changeQuorumVotes***
- ***changeProposalThreshold***
- ***changeVotingPeriod***

- ***propose***
- ***execute***
- ***getAction***
- ***getReceipt***
- ***casteVote***
- ***castVoteBySig***

Recommendation:
The above-mentioned functions should be assigned external visibility.

A. 7 Comparison to boolean Constant

Status: CLOSED

Line no:

- **GovCoordinator.sol - 150**

Description:
Boolean constants can directly be used in conditional statements or require statements. Therefore, it's not considered a better practice to explicitly use **TRUE or FALSE** for comparisons.

Recommendation:
The equality to boolean constants must be eradicated from the above-mentioned line.

A.8 Absence of Error messages in Require Statements

Status: CLOSED

Description:
Some of the ***require statements*** in the contract don't include an error message. While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

Recommendation:
Error messages should be included in every require statement

A.9 NatSpec Annotations must be included

Status: Not Considered

Description:
Smart Contract does not include the NatSpec Annotations adequately.

Recommendations:
Cover by NatSpec all Contract methods.

B. Contract Name - MuteGovernance.sol

Medium Severity Issues

B.1 Require Statements should be used for Zero Address Validation Checks

Line no: 99-117

Status: Not Considered

Description:

The **_moveDelegates** function use if statements to validate the addresses passed by the user.
Since Zero Addresses are completely invalid user input and the function shall not be executed if such addresses are passed, it is considered a better practice to use **require statements** rather than **if statements** for such input validations.

```
 99    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
100        if (srcRep != dstRep && amount > 0) {
101            if (srcRep != address(0)) {
102                // decrease old representative
103                uint32 srcRepNum = numCheckpoints[srcRep];
104                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1]
105                uint256 srcRepNew = srcRepOld.sub(amount);
106                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
107            }
108
109            if (dstRep != address(0)) {
110                // increase new representative
111                uint32 dstRepNum = numCheckpoints[dstRep];
112                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1]
113                uint256 dstRepNew = dstRepOld.add(amount);
114                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
```

Recommendation:

It is recommended to validate the user inputs using **require** statements for the above-mentioned function.

Low Severity Issues

B.2 SafeMath library should be used for arithmetic operations

Status: Not Considered

Line no: 83-94, 104,112,122,126

Description:

The MuteGovernance contract includes some function that doesn't use the Safemath library while performing arithmetic operations like subtraction or division.

Recommendations:
It is recommended to use SafeMath Library while performing any arithmetic operations in the contract.

B.3 NatSpec Annotations must be included

Status: Not Considered

Description:
Smart Contract does not include the NatSpec Annotations adequately.

Recommendations:
Cover by NatSpec all Contract methods.

Informational

B.4 Coding Style Issues

Status: Not Considered

Description:
Coding style issues influence code readability and in some cases may lead to bugs in future. Smart Contracts have a naming convention, indentation and code layout issues.

The order of functions as well as the rest of the code layout does not follow the solidity style guide.

Layout contract elements in the following order:

a. Pragma statements

b. Import statements

c. Interfaces

d. Libraries

e. Contracts

Inside each contract, library or interface, use the following order:

a. Type declarations

b. State variables

c. Events

d. Functions

Recommendations:

Please read the following documentation links to understand the correct order: - https://docs.soliditylang.org/en/v0.6.12/style-guide.html#order-of-functions

---

Closing Summary

Overall, smart contracts are very well written and adhere to guidelines.

All of the critical issues found in the initial audit have now been fixed.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the **Mute platform**. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Mute Team

put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.