Security Audit Report

07/06/2024

Neo X - Governance

RED4SEC

Content

Introduction

The contracts of the **Neo X** system are a set of contracts built into Solidity with predefined addresses. They represent the governance and economic model of **Neo X**, which is completely decentralized and transparent.

The governance contracts of **Neo X** manage both the election of consensus nodes and the distribution of rewards. These contracts are not implemented through transactions, but rather are allocated in the genesis file.

As solicited by **Neo** and as part of the vulnerability review and management process, Red4Sec has been requested to perform a security code audit in order to evaluate the security of the **Neo X - Governance** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of **Neo X Governance Contracts**. The performed analysis shows that the smart contract did contain high risk vulnerabilities.

Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered either "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project, nor as guarantee on the operation or viability of the implemented financial product.

Red4Sec makes full effort and applies every resource available for each audit, however it does not warrant the function, nor the safety of the project and it cannot be deemed a sufficient assessment of the code's utility and safety, bug-free status, or any other declarations of the project. Additionally, Red4Sec makes no security assessments or judgments about the underlying business strategy, or the individuals involved in the project.

Blockchain technology and cryptographic assets come with their own new risks and challenges, where the ecosystem, platform, its programming language, and other software related to said technology can have vulnerabilities that could lead to exploits. As a result, the audit cannot guarantee the explicit security of the audited projects.

The audit reports can be used to improve the code quality of smart contracts, to help limit the vectors of attack and to lower the high level of risks associated with utilizing new and continually changing technologies such as cryptographic tokens and blockchain, but they are unable to detect any future security concerns with the related technologies.

Scope

Red4Sec Cybersecurity has made a thorough audit of the **Neo X Governance Contracts** security level against attacks, identifying possible errors in the design, configuration, or programming; therefore, guaranteeing the availability, integrity and confidentiality of the project and the possible assets treated and stored.

The scope of this evaluation includes the following items provided by **Neo**:

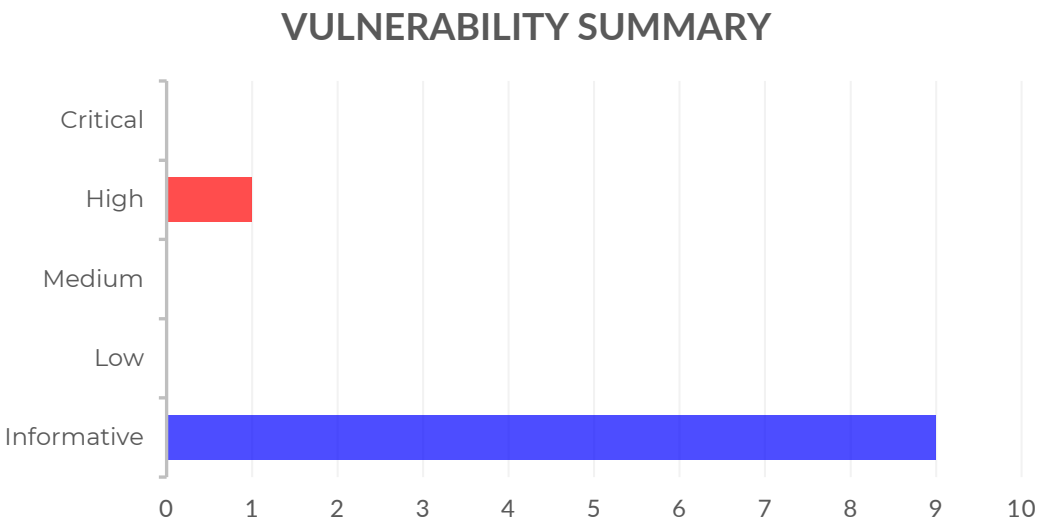
- https://github.com/bane-labs/go-ethereum/tree/governance-change-vote/contracts
    - commit: `ca856ce0ad5d20490d8e7304430f4db3e6331340`

Executive Summary

The security audit against **Neo X Governance** has been conducted between the following dates: **22/04/2024** and **07/06/2024**.

Once the analysis of the technical aspects has been completed, the performed analysis showed that the audited source code did contain high risk vulnerabilities that were mitigated by the team.

During the analysis, a total of **10 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Vulnerabilities Severity annex.



Conclusions

The security audit of the **Neo X Governance** smart contracts have revealed several vulnerabilities that could potentially expose the project to various risks. The most critical vulnerability identified is the High Resource Consumption in the `candidateList`. This issue poses a high risk as it could lead to the exhaustion of resources, negatively affecting the performance and functionality of the smart contract. Therefore, it was strongly recommended to optimize the code to reduce resource consumption.

The **Neo X** team has worked and resolved most of the issues identified and reported in this document, as reflected in their status.

The audit also identified Missing Storage Gaps in Upgradable Contracts which is considered a informative risk. The missing storage gaps could lead to inconsistencies and issues during contract upgrades, while the ability to manipulate reward calculations could adversely impact the fairness and integrity of the contract. It was advised to ensure proper handling and initialization of storage variables and to implement a secure and transparent reward calculation mechanism.

Additional informative risks were also identified, including Uninitialized Governance Variables, GAS Optimizations, Lack of Event Indexing, Code Quality, Optimized Error Reporting, Reentrancy Pattern, Outdated Compiler, and Absence of Unit Tests. While these issues do not pose immediate risks, they could lead to potential problems in the future if not addressed. For instance, uninitialized governance variables could lead to unpredictable contract behavior, while the outdated compiler could result in the contract not benefiting from the latest security updates and features.

To mitigate these risks, it was recommended to create an initialize method in order to set all governance variables, optimize GAS usage, index all events for easy tracking, improve code quality, implement a robust error reporting mechanism, avoid reentrancy patterns, update the compiler to the latest version, and develop comprehensive unit tests to ensure the contract behaves as expected under different scenarios.

In conclusion, the project in its previous form exposed to various vulnerabilities, that were mitigated with some effort. Immediate attention was given to the high and medium risk vulnerabilities, and a thorough review of the development cycle should be conducted to incorporate missing best practices like unit testing and gas optimization.

As noted earlier and confirmed through the review of fixes, we can verify that the **Neo X** team has effectively addressed and resolved the majority of the issues identified and reported in this document.

Vulnerabilities

In this section, you can find a detailed analysis of the vulnerabilities encountered upon the security audit.

List of vulnerabilities

Below, we have gathered a complete list of the vulnerabilities detected by Red4Sec, presented, and summarized in a way that can be used for risk management and mitigation.

Table of vulnerabilities			
ID	**Vulnerability**	**Risk**	**State**
GVN-01	High Resource Consumption in candidateList	**High**	**Fixed**
GVN-02	Missing Storage Gaps in Upgradable Contracts	**Informative**	**Acknowledged**
GVN-03	Uninitialized Governance Variables	**Informative**	**Acknowledged**
GVN-04	GAS Optimizations	**Informative**	**Fixed**
GVN-05	Lack of Event Indexing	**Informative**	**Fixed**
GVN-06	Code Quality	**Informative**	**Fixed**
GVN-07	Optimize Error Reporting	**Informative**	**Fixed**
GVN-08	Reentrancy Pattern	**Informative**	**Fixed**
GVN-09	Outdated Compiler	**Informative**	**Fixed**
GVN-10	Absence of Unit Test	**Informative**	**Fixed**

Vulnerability details

In this section, we provide the details of each of the detected vulnerabilities indicating the following aspects:

- Category
- Active
- Risk
- Description
- Recommendations

High Resource Consumption in candidateList

Identifier	Category	Risk	State
GVN-01	Denial of Service	**High**	**Fixed**

A critical vulnerability related to excessive resource consumption has been identified in the `onPersist` method of the **Neo X** `Governance` contract, which uses the `EnumerableSet` library.

Specifically, the `values()` method of this library is called multiple times: directly on line 313 and via the `_computeConsensus()` function on line 325. This method is known for its high gas demand, as it copies all values of the `candidateList` set in memory, which is described in the OpenZeppelin documentation as potentially expensive.

```
/**
 * @dev Return the entire set in an array
 *
 * WARNING: This operation will copy the entire storage to memory, which can be quite expensive. This is designed
 * to mostly be used by view accessors that are queried without any gas fees. Developers should keep in mind that
 * this function has an unbounded cost, and using it as part of a state-changing function may render the function
 * uncallable if the set grows to a point where copying to memory consumes too much gas to fit in a block.
 */
function values(Bytes32Set storage set) internal view returns (bytes32[] memory) {
```

EnumerableSet.values() warning

The main concern is that repeatedly using `values()` in a method that alters the state of the contract increases the gas cost significantly, especially as the `candidateList` set grows. This increase in gas consumption not only increases operating costs, but also represents a serious security vulnerability.

```
function onPersist() external {
    // NOTE: suppose onPersist always happens at the beginning of every block
    require(msg.sender == SYS_CALL, "side call not allowed");
    // only settle validator reward if there is no epoch change
    IGovReward(GOV_REWARD).withdraw();
    if (block.number < currentEpochStartHeight + epochDuration) return;

    // update tag values
    currentEpochStartHeight = block.number;
    address[] memory candidates = candidateList.values();
    uint length = candidates.length;
    for (uint i = 0; i < length; i++) {
        epochStartGasPerVote[candidates[i]][
            block.number / epochDuration
        ] = candidateGasPerVote[candidates[i]];
    }

    // compute and update consensus
    if (length < consensusSize || totalVotes < voteTargetAmount) {
        currentConsensus = standByValidators;
    } else {
        currentConsensus = _computeConsensus();
    }
    emit Persist(currentConsensus);
}
```

Since the registration of new candidates in `candidateList` is public, an attacker could exploit this feature to register a large number of candidates, causing the execution of `onPersist` to consume more resources than can be provided in a block, effectively blocking the function, and causing a denial of service (DoS).

Recommendations

- Limit the use of `values()` in critical functions: Redesign the `onPersist` method to eliminate repeated use of the `values()` method within operations that change the state of the contract. Consider using iterators or alternative mechanisms that do not require copying the entire set into memory.
- Perform load testing and contract optimization to ensure that key functions can handle a high number of interactions without exceeding resource limits. These tests will help identify and mitigate potential failure points due to excessive gas consumption.

References

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/EnumerableSet.sol#L211-L218

Source Code References

- contracts/solidity/Governance.sol#L313
- contracts/solidity/Governance.sol#L325

Fixes Review

The issue has been fixed in the following pull request:

- https://github.com/bane-labs/go-ethereum/pull/216

Missing Storage Gaps in Upgradable Contracts

Identifier	Category	Risk	State
GVN-02	Design Weaknesses	**Informative**	**Acknowledged**

In **NeoX** governance contracts, the Universal Upgradeable Proxy Standard (UUPS) functionality is used to enable future upgrades. However, it has been observed that the base contracts lack reserved storage variables (`__gap`), a recommended practice in the design of upgradeable contracts. This design omits a crucial component to prevent overwriting of state variables in future contract updates.

Storage gaps serve as a convention to allocate storage spaces within a base contract, ensuring future versions can use these spaces without impacting the storage layout of child contracts.

The lack of reserved storage variables may lead to storage conflicts when new state variables are introduced in future contract versions. Solidity's state storage layout is sequential and rigid, and any new variables will be aligned directly after the last existing ones. Without reserved storage spaces (`__gap`), there is a risk of overwriting existing memory locations.

Recommendations

- Introduction of `__gap` variables: Modify base contracts to include reserved `__gap` storage variables, appropriately sized to allow future expansion of the contract without affecting existing state variables. These variables act as buffers between variable definitions to prevent unintentional overwrites.
- Maintain detailed and up-to-date documentation of the contract storage structure, including the position and purpose of each variable and the reserved `__gap` spaces. This will make it easier to safely manage future updates and help avoid critical errors in storage layout.

References

- https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#storage-gaps

Source Code References

- contracts/solidity/GovernanceVote.sol#L8

Fixes Review

The Neo team informed the following:

```
Considering the contract storage compatibility with deployed NeoX T3, this change is
not planned.
```

Uninitialized Governance Variables

Identifier	Category	Risk	State
GVN-03	Design Weaknesses	**Informative**	**Acknowledged**

The presence of multiple uninitialized critical variables has been detected in the `Governance.sol` contract, which are part of the **NeoX** governance infrastructure. These variables (`consensusSize`, `minVoteAmount`, `voteTargetAmount`, `registerFee` and `epochDuration`) are essential for the operability and correct functioning of the governance system. The absence of initialization and methods for their configuration results in these values remaining zero, which means that the governance system will not function as intended.

It has been considered to eliminate this issue due to the operation of **Neo X** itself, where the mentioned values are initialized at the genesis of the blockchain. However, because the contract code can be reused, or even updated afterwards, it has been decided to be kept as a briefing note for the development team. Furthermore, being able to establish these values would facilitate the creation of unit tests.

Contracts must have a `constructor` in which these values are established, or in the case of updateable contracts, an `initialize` method that fulfills this function.

The lack of appropriate values in these variables not only prevents the normal execution of governance functions, but also exposes the system to significant security and functionality risks. For example, without an appropriate `minVoteAmount`, votes could be recorded without minimum participation guarantees, affecting the legitimacy of the decisions made. Similarly, an `epochDuration` not properly configured could result in irregular or indefinite voting periods, disrupting the entire governance cycle.

Recommendations

- **Initialization during deployment:** Modify the contract to include a constructor or in the case of updatable contracts, an initialization function that sets safe and reasonable default values for all critical variables during contract deployment. This ensures that the contract begins its operation with a defined and predictable state.
- **Secure configuration methods:** Implement administrative functions that allow the configuration of these post-deployment variables. These features should be protected by access controls so that only authorized users (such as administrators or through a voting mechanism) can modify them.
- **Integrity validations:** Incorporate checks into functions that depend on these values to ensure that they operate within expected parameters. For example, verify that `epochDuration` and `minVoteAmount` are not zero before proceeding with voting and consensus calculation operations.

Source Code References

- contracts/solidity/Governance.sol#L65-L73

Fixes Review

The Neo team informed the following:

> They are initialized in the genesis block allocation. Plz refer to https://github.com/bane-labs/go-ethereum/blob/bane-main/config/genesis_testnet.json.
>
> The codes of these system contracts are not deployed but directly setted, so constructors will not be executed.

GAS Optimizations

Identifier	Category	Risk	State
GVN-04	Codebase Quality	**Informative**	**Fixed**

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On EVM blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard-coded constant instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to arithmetical operations and comparisons.

Avoid Compound Assignment Operator

It is recommended to avoid using compound assignment operators (e.g., `+=`, `-=`, `*=`, etc.) on state variables. Compound assignment operators in Solidity require multiple steps to execute and can result in increased gas costs compared to simple assignment, like `var = var + x`.

By avoiding compound assignment operators and using simple assignment instead, it is possible to reduce gas costs, making the project more efficient.

Source Code References

- contracts/solidity/Governance.sol#L143
- contracts/solidity/Governance.sol#L170
- contracts/solidity/Governance.sol#L229-L231
- contracts/solidity/Governance.sol#L281
- contracts/solidity/Governance.sol#L419

Logic Optimization

The `_computeReward` method initializes the local variables `height`, `lastGasPerVote` and `latestGasPerVote`. However, it immediately follows with a conditional statement. If this condition is not met, the method returns 0 directly without using the previously declared variables.

```
function _computeReward(
    address voter,
    address candidate
) internal view returns (uint) {
    // NOTE: suppose onPersist always happens at the beginning of every block,
    uint height = voteHeight[voter];
    uint lastGasPerVote = voterGasPerVote[voter];
    uint latestGasPerVote = candidateGasPerVote[candidate];
    if (currentEpochStartHeight <= height) return 0;
```

It is advisable to perform the check before declaring all the variables, in order to optimize the gas consumption of the method as much as possible in the event that the Epoch has not finished.

Source Code References

- contracts/solidity/Governance.sol#L340-L341

In the `registerCandidate` and `exitCandidate` methods, the contains method of the candidateList set is used to check whether a candidate already exists before adding or removing it. By reusing the `boolean` return from the call to the `add` and `remove` methods, which returns whether it was processed correctly. Therefore, it is possible to save gas by eliminating the `contains` check.

```
function registerCandidate(uint shareRate) external payable {
    require(tx.origin == msg.sender, "only allow EOA");
    require(msg.value >= registerFee, "insufficient amount");
    require(shareRate < 1000, "invalid rate");
    require(!candidateList.contains(msg.sender), "candidate exists");
    require(exitHeightOf[msg.sender] == 0, "left not claimed");
    candidateList.add(msg.sender);
    if (receivedVotes[msg.sender] > 0) {
        totalVotes += receivedVotes[msg.sender];
    }

    // record share rate and balance
    shareRateOf[msg.sender] = shareRate;
    candidateBalanceOf[msg.sender] = msg.value;
    emit Register(msg.sender);
}
```

Source Code References

- contracts/solidity/Governance.sol#L168
- contracts/solidity/Governance.sol#L182

The `checkVote` function performs a full run over `voters` to count how many have voted. The problem is that the `for` loop continues even after the necessary number of votes has already been reached in order to consider the vote successful.

```
function checkVote(
    bytes32 methodKey,
    bytes32 paramKey
) internal view returns (bool isPass) {
    address[] memory voters = IGovReward(govReward).getMiners();
    uint votedCount;
    for (uint i; i < voters.length; i++) {
        if (voteMap[methodKey][voters[i]] == paramKey) {
            votedCount++;
        }
    }
    return votedCount >= (voters.length + 1) / 2;
}
```

It is advisable to introduce a conditional inside the `for` loop in order to stop its execution as soon as the necessary threshold of votes has been reached. This not only reduces gas consumption but also optimizes the execution time of the function.

Source Code References

- contracts/solidity/GovernanceVote.sol#L53

Use Constants in needVote

It has been observed that calls to the `needVote` modifier redundantly calculate hashes with static values, resulting in excessive gas consumption during transactions.

```
// add blacklist
function addBlackList(
    address _addr
)
    external
    needVote(keccak256("addBlackList"), keccak256(abi.encode(_addr)))
{
    require(!isBlackListed[_addr], "Policy: Blacklist already exists");
    isBlackListed[_addr] = true;
    emit AddBlackList(_addr);
}

// remove blacklist
function removeBlackList(
    address _addr
)
    external
    needVote(keccak256("removeBlackList"), keccak256(abi.encode(_addr)))
{
    require(isBlackListed[_addr], "Policy: Blacklist does not exist");
    delete isBlackListed[_addr];
    emit RemoveBlackList(_addr);
}
```

It is recommended to previously calculate the hash values of these constant values within the contract. Therefore, these values can be accessed without the need to recalculate them on each call to the modifier, thus reducing gas consumption and improving the overall efficiency of the contract.

Source Code References

- contracts/solidity/Policy.sol#L55
- contracts/solidity/Policy.sol#L67
- contracts/solidity/Policy.sol#L79
- contracts/solidity/Policy.sol#L94

Duplicate External Contract Calls

An inefficiency has been detected in the contract's `needVote` modifier, which makes three calls to the `getMinners` function through different methods. This practice results in redundant use of resources which triggers an increase in gas consumption.

Source Code References

- contracts/solidity/GovernanceVote.sol#L56

Increase Operation Optimization

During the audit it was found that it is possible to optimize all the `for` loops in the project.

There are two main ways to increment a variable in solidity:

```
++i will increment the value of i, and then return the incremented value.
i++ will increment the value of i, but return the original value that i held before
being incremented.
```

Since the return of the increment operation of the `for` loop is indifferent and discarded, both `i++` or `++i` instructions are completely valid instructions. However, the `++i` instruction has a considerably lower cost compared to incrementing a variable using `i++`, for this reason it is convenient to use `++i` for the increments of the for loops.

An example of this behavior can be seen below:

```
receive() external payable nonReentrant {
    require(msg.sender == GOV_REWARD, "side call not allowed");
    address[] memory validators = currentConsensus;
    uint length = validators.length;
    for (uint i = 0; i < length; i++) {
        if (receivedVotes[validators[i]] != 0) {
```

References

- https://docs.soliditylang.org/en/latest/types.html#compound-and-increment-decrement-operators

Source Code References

- contracts/solidity/GovernanceVote.sol#L22
- contracts/solidity/GovernanceVote.sol#L37
- contracts/solidity/GovernanceVote.sol#L48
- contracts/solidity/GovernanceVote.sol#L50
- contracts/solidity/Governance.sol#L141

- contracts/solidity/Governance.sol#L377
- contracts/solidity/Governance.sol#L386
- contracts/solidity/Governance.sol#L315
- contracts/solidity/Governance.sol#L398
- contracts/solidity/Governance.sol#L401
- contracts/solidity/Governance.sol#L419

Caching Arrays Length in Loops

Every time an array iteration occurs in a loop, the solidity compiler will read the array's length. Thus, this is an additional `sload` operation for storage arrays (100 more extra gas for each iteration except for the first) and an additional `mload` action for memory arrays (3 additional gas for each iteration except for the first).

```
for (uint i; i < array.length; ++i) {
    // ...
}
```

The array length (in stack) might be cached to reduce these additional costs:

```
uint length = array.length;
for (uint i; i < length; ++i) {
    // ...
}
```

The `sload` or `mload` action is only performed once in the example above before being replaced by the inexpensive `dupX` instruction.

Source Code References

- contracts/solidity/GovernanceVote.sol#L22
- contracts/solidity/GovernanceVote.sol#L37
- contracts/solidity/GovernanceVote.sol#L48

Dead Code

In programming, a part of the source code that is never used is known as dead code. The execution of this type of code consumes more GAS during deployment, something that is not necessary.

The `safeTransfer` method of the `TransferHelper` library is not used during the execution of the contract, so it would be convenient to either remove it or to use it.

Source Code References

- contracts/solidity/GovReward.sol#L7

Fixes Review

The issue has been addressed in the following pull request:

- https://github.com/bane-labs/go-ethereum/pull/195

Lack of Event Indexing

Identifier	Category	Risk	State
GVN-05	Auditing and Logging	**Informative**	**Fixed**

A significant omission has been detected in event indexing in the **NeoX** governance smart contracts. Events such as `Vote`, `Register`, `Exit`, `Revoke`, `AddBlackList`, `RemoveBlackList`, among others, are essential for decentralized applications (DApps) and in effectively processing off-chain data. However, the key parameters of these events are not indexed, significantly complicating the search and filtering process based on specific attributes such as user addresses.

The absence of event indexing in the `GovernanceVote`, `Policy`, and `Governance` contracts obstruct the efficient filtering of these events by address, a standard practice in DApp development. This limitation not only diminishes operational efficiency when engaging with these contracts but also inflates development costs and timelines by complicating the integration and real-time monitoring of events.

Recommendations

- Index relevant event parameters: Modify the declaration of all critical events to include indexing of key parameters. For example, in the `Vote` event, it should be changed to `event Vote(address indexed voter, bytes32 methodKey, bytes32 paramKey)`. This will allow developers to filter events by voter address, making it easier to search and analyze events.
- Perform a thorough review of all events in the contracts to ensure that all relevant parameters are properly indexed.

Source Code References

- contracts/solidity/GovernanceVote.sol#L10
- contracts/solidity/Policy.sol#L16-L17

  contracts/solidity/Governance.sol#L11-L13

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/bane-labs/go-ethereum/pull/196

Code Quality

Identifier	Category	Risk	State
GVN-06	Codebase Quality	**Informative**	**Fixed**

It has been possible to verify that, despite the good quality of the code, there is a lack of order and structure that makes reading and analyzing the code difficult. This is a bad practice, especially in these types of projects that are continually changing and improving.

High-quality code is important for various reasons, including maintainability, reliability, efficiency, scalability, and security. Well-written code that adheres to best practices offers several advantages, such as easier to maintain and extend, less prone to errors, more efficient, improved collaboration potential, scalability, and heightened security. Ultimately, good code quality contributes to a more dependable, maintainable, and secure software product, improving the overall user experience while reducing the risk of introducing new vulnerabilities.

Improvable Dependencies

It has been noted that the `ERC1967Proxy` library has been included directly in the project code instead of being managed as an external dependency. This approach increases contract size and can lead to code duplication, especially if multiple contracts within the same project use the same library. In addition, it complicates the updating and maintenance of the code since any change in the library requires a recompilation and a new deployment of the affected contracts.

Source Code References

- contracts/solidity/ERC1967Proxy.sol

The `Governance.sol` contract implements the `ReentrancyGuard` dependency of the OpenZeppelin standard library in its non-updatable version. For contracts designed to be upgradeable, it is recommended to use `@openzeppelin/contracts-upgradeable`, which includes a version of `ReentrancyGuard` specifically designed to be used with upgradeable proxies.

Source Code References

- contracts/solidity/Governance.sol#L5

Unsafe Cast

In the `_topK function`, a value of type `int` is cast to `uint` in an unsafe manner. This cast can lead to unwanted behavior or errors at runtime, especially if the value of `int` is negative, in case the contract is incorrectly initialized.

```
function _topK(
    address[] memory candidates,
    uint[] memory votes,
    uint k
) internal pure {
    uint length = candidates.length;
    for (int j = int(k) / 2 - 1; j >= 0; j--) {
        _heapDown(candidates, votes, uint(j), k);
    }
    for (uint i = k; i < length; i++) {
        if (votes[i] > votes[0]) {
            votes[0] = votes[i];
            candidates[0] = candidates[i];
            _heapDown(candidates, votes, 0, k);
        }
    }
}
```

Source Code References

- contracts/solidity/Governance.sol#L399

Avoid Duplicate Code

The `_safeTransferETH` method of the `Governance` contract is already included in the existing `TransferHelper` library in the `GovReward.sol` file. Therefore, it is convenient to reuse the library, and avoid duplicating code, which will make it more complicated to maintain.

Source Code References

- contracts/solidity/Governance.sol#L367

Use of Selector

In the current `safeTransfer` function, a hard-coded function selector (`0xa9059cbb`) is used to interact with the `transfer` method. Although this is functional, it does not provide transparency into the operation being performed, making the code difficult to read, maintain, and audit.

```
function safeTransfer(address token, address to, uint256 value) internal {
    // bytes4(keccak256(bytes('transfer(address,uint256)')));
    (bool success, bytes memory data) = token.call(
        abi.encodeWithSelector(0xa9059cbb, to, value)
    );
    require(
        success && (data.length == 0 || abi.decode(data, (bool))),
        "safeTransfer: transfer failed"
    );
}
```

Using hard-coded selectors can make the code less readable and more difficult to verify, especially for people who are not familiar with the specifics of function selectors. To improve code clarity and maintainability, it is recommended to use `IERC20.transfer.selector`, which automatically generates the function selector based on the interface declaration.

References

- https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/TransferHelper.sol#L34

Source Code References

- contracts/solidity/GovReward.sol#L10

Improvable Event Emit

The `_settleReward` function emits the `VoterClaim` event every time a reward is calculated and assigned, regardless of whether the reward value is zero. This can lead to unnecessary emission of events, which not only consumes gas but can also generate noise in the event logs, complicating data analysis and contract monitoring.

```
function _settleReward(
    address voter,
    address candidate
) internal returns (uint) {
    uint reward = _computeReward(voter, candidate);
    voterGasPerVote[voter] = candidateGasPerVote[candidate];
    emit VoterClaim(voter, reward);
    return reward;
}
```

It is recommended to modify the `_settleReward` function so that it only raises the `VoterClaim` event when the reward is non-zero. This change ensures that events are only emitted when there is a significant status update, optimizing gas usage and improving log interpretation.

Source Code References

- contracts/solidity/Governance.sol#L363

Solidity Literals

Although the use of exponential literals is technically correct and does not directly affect contract performance, it can compromise how the code is read and understood, in order to make the code easier to read and to minimize human errors. Solidity recommends the use of literals which consequently makes it more user friendly. Additionally, readability is crucial to avoid errors and facilitate code auditing and validation.

```
uint public constant SCALE_FACTOR = 10 ** 18;
```

Source Code References

- contracts/solidity/Governance.sol#L63

Improvable Contract Structure

Interfaces and libraries are currently being included within the contracts themselves instead of keeping them in separate files. This mixture of contract and interface definitions in a single file can complicate the maintainability and clarity of the code, which is why it is a totally discouraged practice.

Source Code References

- contracts/solidity/GovernanceVote.sol#L4-L6
- contracts/solidity/GovReward.sol#L6-L33
- contracts/solidity/Governance.sol#L8-L50

Improve Nomenclature on Internal Methods

In Solidity, it is common practice to use an `underscore (_)` at the beginning of internal function names to distinguish them from public or external functions. This helps clarify feature visibility and ensures that code is more readable and maintainable, particularly in large and complex contracts.

```
function vote(bytes32 methodKey, bytes32 paramKey) internal {
    voteMap[methodKey][msg.sender] = paramKey;
    emit Vote(msg.sender, methodKey, paramKey);
}

function clearVote(bytes32 methodKey) internal {
    address[] memory voters = IGovReward(govReward).getMiners();
    for (uint i; i < voters.length; i++) {
        delete voteMap[methodKey][voters[i]];
    }
}
```

Adopting this naming convention does not directly impact contract performance or gas consumption, but it significantly improves code readability and reduces the risk of errors, especially in the integration and maintenance of extensive code.

Source Code References

- contracts/solidity/GovernanceVote.sol#L30
- contracts/solidity/GovernanceVote.sol#L35
- contracts/solidity/GovernanceVote.sol#L42

Named Mappings

Solidity 0.8.18 allows to use named parameters in mapping types. This new feature enables the assignment of descriptive names to mappings, improving the code's readability, interpretation, and audit.

Reference

- https://soliditylang.org/blog/2023/02/01/solidity-0.8.18-release-announcement/

Source Code References

- contracts/solidity/GovernanceVote.sol#L18
- contracts/solidity/Policy.sol#L12
- contracts/solidity/Governance.sol#L78-L106

Recommendations

As a reference, it is always advisable to apply coding style and good practices, which can be found in multiples standards such as:

- Solidity Style Guide

These references are very useful to improve the quality of the smart contract. A few of these practices are generally known and accepted forms to develop software.

Fixes Review

The Neo team has addressed part of the issue in the following pull request and is aware of the remaining potential optimizations.

- https://github.com/bane-labs/go-ethereum/pull/217

Optimize Error Reporting

Identifier	Category	Risk	State
GVN-07	Auditing and Logging	**Informative**	**Fixed**

The project has adequate error handling and clear error messages. However, the system for reporting errors can be optimized by using the improvements introduced in the latest solidity versions for a more efficient form of notifying users about a failure in the operation of the project and lower GAS consumption.

Reduce Require Messages Length

Ethereum Virtual Machine operates under a 32-byte word memory model where an additional gas cost is paid by any operation that expands the memory that is in use.

Therefore, exceeding error messages of this length means increasing the number of slots necessary to process the require, reducing the error messages to 32 bytes or less would lead to saving gas.

Source Code References

- contracts/solidity/Policy.sol#L84
- contracts/solidity/Policy.sol#L96
- contracts/solidity/GovReward.sol#L20
- contracts/solidity/Governance.sol#L369

Use Custom Errors Instead of Require

Custom errors are more gas efficient than revert strings in terms of deployment and runtime cost when the revert condition is met. The `require` statement is more expensive than the use of custom errors because it requires placing the error message on the stack, whether or not the transaction is reverted and regardless of the result of the condition.

Solidity **0.8.4** introduced custom errors, a more efficient way to notify users of an operation failure, The new custom errors are defined through the `error` statement, and they can also receive arguments. As indicated in the following illustrative example:

```
error Unauthorized();
error InsufficientPrice(uint256 available, uint256 required);
```

Afterwards, it is enough to just call them when needed using `if` conditionals:

```
if (msg.sender != owner) revert Unauthorized();
if (amount > balance[msg.sender]) {
    revert InsufficientPrice({
        available: balance[msg.sender],
        required: amount
    });
```

This behavior has been observed in the contracts listed below.

Source Code References

- contracts/solidity/GovernanceVote.sol#L57
- contracts/solidity/Policy.sol#L22
- contracts/solidity/Policy.sol#L57
- contracts/solidity/Policy.sol#L69
- contracts/solidity/Policy.sol#L84
- contracts/solidity/Policy.sol#L96
- contracts/solidity/GovReward.sol#L12
- contracts/solidity/GovReward.sol#L20
- contracts/solidity/GovReward.sol#L45
- contracts/solidity/GovReward.sol#L50
- contracts/solidity/Governance.sol#L109
- contracts/solidity/Governance.sol#L138
- contracts/solidity/Governance.sol#L163-L167
- contracts/solidity/Governance.sol#L180
- contracts/solidity/Governance.sol#L193
- contracts/solidity/Governance.sol#L210-L213
- contracts/solidity/Governance.sol#L244
- contracts/solidity/Governance.sol#L274
- contracts/solidity/Governance.sol#L293
- contracts/solidity/Governance.sol#L306
- contracts/solidity/Governance.sol#L369

References

- https://blog.soliditylang.org/2021/04/21/custom-errors

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/bane-labs/go-ethereum/pull/209

Reentrancy Pattern

Identifier	Category	Risk	State
GVN-08	Timing and State	**Informative**	**Fixed**

The Reentrancy attack is a vulnerability that occurs when external contract calls can make new calls to the calling contract before the initial execution is completed. For a function, this means that the state of the contract could change in the middle of its execution as a result of a call to an untrusted contract or the use of a low-level function with an external address.

One way to mitigate this vulnerability is to use a checks-effects-interactions pattern in the contract design. In this pattern, the contract first performs all the necessary checks to ensure that the function call is valid and then performs all the state changes before interacting with other contracts or accounts.

In the case of the `needVote` modifier of the `GovernanceVote` contract, votes are cleared after external calls are made, so the call can be redirected to any method before updating the values.

```
modifier needVote(bytes32 methodKey, bytes32 paramKey) {
    require(isMiner(msg.sender), "not Miner");
    // update vote map
    vote(methodKey, paramKey);
    // check vote, if not pass just return
    if (!checkVote(methodKey, paramKey)) {
        return;
    }
    // execute method
    _;
    emit VotePass(methodKey, paramKey);
    // clear vote
    clearVote(methodKey);
}
```

By applying the security measures described, the contract can ensure that it completes all state changes before allowing any external interactions, preventing reentrancy and other types of attacks in smart contracts.

Recommendations

It is **essential to always make the state changes in the storage before making transfers or calls to external contracts**, in addition to implementing the necessary measures to avoid duplicated calls or chain calls to the methods. A good practice to avoid reentrancy is to organize the code according to the following pattern:

- The first step is conducting all necessary **checks**.
- The next step is the application of all **effects.**
- All external **interactions** take place in the last step.

References

- https://docs.soliditylang.org/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern
- https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

Source Code References

contracts/solidity/GovernanceVote.sol#L66-L68

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/bane-labs/go-ethereum/pull/195

Outdated Compiler

Identifier	Category	Risk	State
GVN-09	Outdated Software	**Informative**	**Fixed**

Solc frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect this version. The project has set the `^0.8.20` compiler version in the config.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
```

Governance.sol pragma

The Solidity branch up to version `0.8.25`, similar to version `0.8.22`, has incorporated significant bug fixes in the updates. Consequently, it is recommend adopting the latest pragma version to ensure optimal functionality and address any potential issues.

Recommendations

- It is always a good policy to use the most up to date version of the pragma.

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md

Source Code References

- contracts/solidity/Governance.sol#L2
- contracts/solidity/GovernanceVote.sol#L2
- contracts/solidity/GovProxyAdmin.sol#L2
- contracts/solidity/GovReward.sol#L2

  contracts/solidity/Policy.sol#L2

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/bane-labs/go-ethereum/pull/208

Absence of Unit Test

Identifier	Category	Risk	State
GVN-10	Testing and Documentation	**Informative**	**Fixed**

During the security review, we identified the absence of Unit Tests, a best practice that is not only highly recommended but has also become mandatory for projects managing large amounts of capital.

For the safety development of any project, at Red4Sec we consider that unitary tests are essential, and its periodical execution is a fundamental practice.

Unit tests play a crucial role for the following reasons:

- **Early error detection:** It assists in identifying and rectifying errors at an early stage of the software development process.
- **Enhanced code quality**: By confirming that each unit operates correctly in isolation, unit tests contribute to improved overall code quality.
- **Problem identification and resolution:** It facilitates pinpointing and resolve issues, both current and future.
- **Foundation for new features:** Unit tests establish a solid foundation for the development of new features, ensuring its proper integration.
- **Confidence in code:** it increases the confidence in the code, allowing for changes and updates with the assurance that existing functionalities remain working.
- **Simplified integration:** Unit tests facilitate seamless integration and enhance the collaboration process within development teams.

Fixes Review

This issue has been addressed in the following pull request:

- https://github.com/bane-labs/go-ethereum/pull/195

Annexes

Methodology

A code audit is a thorough examination of the source code of a project with the objective of identifying errors, discovering security breaches, or contraventions of programming standards. It is an essential component to the defense in programming, which seeks to minimize errors prior to the deployment of the product.

Red4Sec adopts a set of cybersecurity tools and best security practices to audit the source code of the smart contract by conducting a search for vulnerabilities and flaws.

The audit team performs an analysis on the functionality of the code, a manual audit, and automated verifications, considering the following crucial features of the code:

- The implementation conforms to protocol standards and adheres to best coding practices.
- The code is secure against common and uncommon vectors of attack.
- The logic of the contract complies with the specifications and intentions of the client.
- The business logic and the interactions with similar industry protocols do not contain errors or lead to dangerous situations to the integrity of the system.

In order to standardize the evaluation, the audit is executed by industry experts, in accordance with the following procedures:

Manual Analysis

- Manual review of the code, line-by-line, to discover errors or unexpected conditions.
- Assess the overall structure, complexity, and quality of the project.
- Search for issues based on the SWC Registry and known attacks.
- Review known vulnerabilities in the third-party libraries used.
- Analysis of the business logic and algorithms of the protocol to identify potential risk exposures.
- Manual testing to verify the operation, optimization, and stability of the code.

Automatic Analysis

- Scan the source code with static and dynamic security tools to search for known vulnerabilities.
- Manual verification of all the issues found by the tools and analyzes their impact.
- Perform unit tests and verify the coverage.

Vulnerabilities Severity

Red4Sec determines the severity of vulnerabilities found in risk levels according to the impact level defined by CVSSv3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST), classifying the risk of vulnerabilities on the following scale:

Severity	Description
Critical	Vulnerabilities that possess the highest impact over the systems, services and/or sensitive information. The existence of these vulnerabilities is dangerous and should be fixed as soon as possible.
High	Vulnerabilities that could compromise severely compromise the service or the information it manages even if the vulnerability requires expertise to be exploited.
Medium	Vulnerabilities that on their own can have a limited impact and/or that combined with other vulnerabilities could have a greater impact.
Low	These vulnerabilities do not suppose a real risk for the systems. Also includes vulnerabilities which are extremely hard to exploit or whose impact on the service is low.
Informative	It covers various characteristics, information or behaviors that can be considered as inappropriate, without being considered as vulnerabilities by themselves.

Invest in Security, invest in your future