Security Review Report
NM-0405 GaiaNet AI
(Gaia Token)

NETHERMIND
SECURITY

(Feb 05, 2025)

Contents

1 Executive Summary

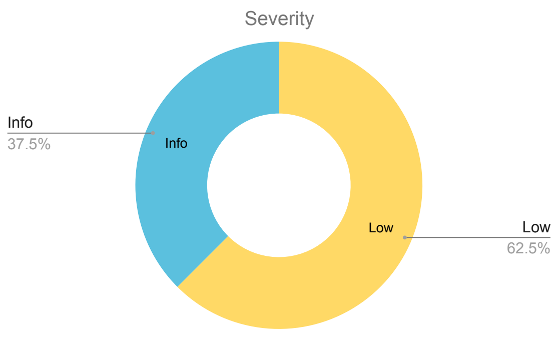
This document outlines the security review conducted by Nethermind Security for Gaia Network. GaiaNet is a decentralized platform that enables users to create, scale, and monetize AI agents tailored to their expertise. Each node includes a fine-tuned LLM, knowledge embedding, a vector database, API server, and plugin system, allowing deployment as virtual counterparts.

This review focuses on analyzing the contracts associated with the Gaia token and governance with time-lock. The team performed a review of 182 lines of code. Throughout the process, the `GaiaNet` team offered clarifications through several meetings and detailed documentation, including flow diagrams that illustrate the interactions between contracts. **The audit was performed using**: (a) manual analysis of the codebase, and (b) writing test cases. **Throughout this document, we identify** 8 points of attention, classified as follows: five as `Low` and three as `Informational`.The issues are summarized in Fig. 1.

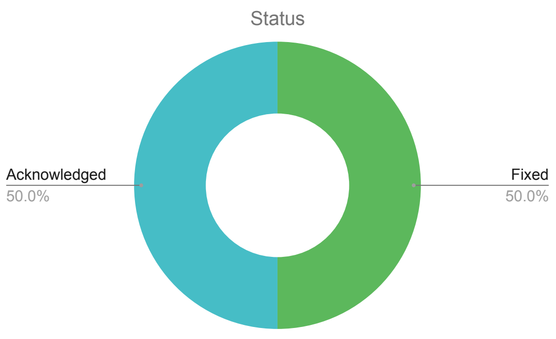
Remarks about GaiaNet documentation

The test analysis for the reviewed contracts reveals gaps in coverage. The `GaiaGovernor` contract lacks tests capable of detecting business logic bugs, relying instead on setter-based tests. Similarly, the `GaiaToken` contract demonstrates **low test coverage**. Comprehensive test coverage is essential to ensure the reliability and security of smart contracts.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the overview of the system. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (5), **Undetermined** (0), **Informational** (3), **Best Practices** (0). **(b) Distribution of status: Fixed** (4), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan 24, 2025
Final Report	Feb 05, 2025
Methods	Manual Review, Tests
Repository	gaia-smart-contracts
Commit Hash - Token and Governance	88de70c18d035d700b63fcc2bed5cb2017f1aaa0
Final Commit Hash	c4afda4d0b0b6a8a156443cf706e521b4068f436
Documentation	Documentation, diagrams, and communication
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	GaiaGovernor.sol	126	7	5.6%	13	146
2	GaiaTimelock.sol	10	13	130.0%	2	25
3	GaiaToken.sol	46	1	2.2%	7	54
	Total	**182**	**21**	**11.5%**	**22**	**225**

3 Summary of Issues

	Finding	Severity	Update
1	Cancellation in GaiaTimelock can't be called from the GaiaGovernor	Low	Acknowledged
2	Safeguarding Contract Operations by Disabling Ownership Renouncement	Low	Fixed
3	Signers cannot cancel permit signatures before the deadline in `GaiaToken`	Low	Fixed
4	`GaiaToken` cannot be paused	Low	Fixed
5	`GaiaToken` extends `Pausable` but does not add pausable functionality	Low	Fixed
6	Quorum value may be incorrect because of bridging	Info	Acknowledged
7	Important deployment information	Info	Acknowledged
8	`GaiaTimelock` could allow anyone to execute proposals	Info	Acknowledged

4 System Overview

This section presents an overview of the *GaiaToken*, *GaiaTimelock*, and *GaiaGovernor* contracts.

4.1 GaiaToken - ERC20 Token for Governance

The **GaiaToken** contract is an ERC20 token that serves as the backbone of the Gaia ecosystem, providing both governance power and economic utility. Based on OpenZeppelin implementations, it includes additional functionalities such as token burning, voting, pausing, and permit-based approvals.

Core Functionalities:

- *Token Burning (`ERC20Burnable`):* Holders can burn tokens, reducing the circulating supply.
- *Permit-Based Approvals (`ERC20Permit`):* Allows users to approve transactions via signatures, reducing gas costs.
- *Governance and Voting (`ERC20Votes`):* Tokens represent voting power, and users can delegate their votes without transferring tokens.
- *Ownership Control (`Ownable`):* Provides an access control mechanism where an account can be granted exclusive access to particular functions.
- *Pausability (`Pausable`):* Allows the contract owner to pause token transfers for instance.

Tokenomics and Allocation:

- *Total supply:* 1,000,000,000 GAIA tokens.
- *Mining allocation:* 40% of the total supply (400,000,000 GAIA) is reserved for mining and rewards.
- *Minting:* All tokens are minted at deployment and assigned to the deployer.

Main public/external functions:

- *Token Transfer*:
  - Users can transfer tokens via the standard ERC20 `transfer()` function.
  - With `ERC20Permit` tokens can be transferred in a single transaction via `permit` function.
  - Transfers automatically update voting power in governance via `ERC20Votes`.
- *Token Burning*:
  - Token holders can burn their tokens using the `burn()` function from `ERC20Burnable`, reducing the circulating supply.
- *Delegation for Governance*:
  - A user can delegate their voting power to another address (or themselves) using `delegate()`.
  - Delegation does not transfer tokens, only voting rights.
- *Pausing Token Transfers (Emergency Feature)*:
  - The owner can pause transfers using the `pause()` function from `Pausable`.
  - When paused, no tokens can be transferred until `unpaused`.

4.2 GaiaTimelock - Secure Execution of Governance Proposals

The `GaiaTimelock` contract introduces a delay between a governance decision and its execution, ensuring security and transparency. It extends OpenZeppelin's `TimelockController`, inheriting access control and execution mechanisms.

Core Functionalities:

- *Governance Execution Delay:*
  - Approved proposals must wait for a minimum delay before execution, preventing instant changes and allowing stakeholders to review decisions.
- *Proposers & Executors:*
  - *Proposers:* Addresses that can create proposals.
  - *Executors:* Addresses that execute approved proposals.
  - These roles are configurable through governance.
- *Admin Role:*
  - The admin can manage proposers and executors to oversee governance execution.
- *Standardized Governance (TimelockController):*
  - Ensures compatibility with other governance modules.

Constructor Parameters:

The code below presents the parameters received by the constructor:

```
    constructor(
        uint256 minDelay,
        address[] memory proposers,
        address[] memory executors,
        address admin
    ) TimelockController(minDelay, proposers, executors, admin) {}
```

The values set during deployment are:

a) *minDelay* – Minimum delay before execution.

b) *proposers* – Addresses with the right to create proposals.

c) *executors* – Addresses authorized to execute approved proposals.

d) *admin* – The initial administrator.

4.3 GaiaGovernor - On-Chain Voting & Proposal Execution

The `GaiaGovernor` contract is the central governance mechanism of the Gaia network, enabling token holders to propose, vote on, and execute decisions. It extends OpenZeppelin's Governor contracts, making governance transparent and decentralized.

Core Functionalities:

- *Proposal & Voting System* (`Governor`)*:*
  - Token holders can create proposals and vote.
  - A voting delay ensures time before voting starts.
  - If quorum and votes are met, the proposal is approved for execution.
- *Custom Governance Settings* (`GovernorSettings`)*:*
  - Defines the voting delay, the voting period, and proposal threshold.
- *Simple Vote Counting* (`GovernorCountingSimple`)*:*
  - Extension of Governor, considering three options for vote counting: Against, For, and Abstain.
- *Token-Based Voting* (`GovernorVotes`)*:*
  - Extracts voting weight from an `ERC20Votes` token.
- *Quorum Requirement* (`GovernorVotesQuorumFraction`)*:*

- Requires a minimum percentage of votes for proposals to pass.

– *Timelock-Controlled Execution* (`GovernorTimelockControl`)*:*

  - Approved proposals are queued in the `GaiaTimelock` before execution.
  - Implements proposal queuing, execution, and cancellation mechanisms.

The code below presents the parameters received by the constructor:

```
    constructor(
        IVotes _token,
        TimelockController _timelock,
        uint48 _votingDelay,
        uint32 _votingPeriod,
        uint256 _proposalThreshold,
        uint256 _quorumPercentage
    )
        Governor("GaiaGovernor")
        GovernorSettings(_votingDelay, _votingPeriod, _proposalThreshold)
        GovernorVotes(_token)
        GovernorVotesQuorumFraction(_quorumPercentage)
        GovernorTimelockControl(_timelock)
    {}
```

The values set in these parameters represent:

a) *_token* – `GaiaToken` token used for voting.

– *_timelock* – `GaiaTimelock` contract for managing execution delays.

b) *_votingDelay* – Delay before voting begins.

c) *_votingPeriod* – Duration of the voting period.

d) *_proposalThreshold* – Minimum tokens required to submit a proposal.

e) *_quorumPercentage* – Required participation rate for governance approval.

Overridden Functions: The contract overrides multiple functions to integrate different governance extensions:

– *Voting Mechanics:*

  - `votingDelay()` – Returns voting delay time.
  - `votingPeriod()` – Returns the voting duration.
  - `proposalThreshold()` – Defines the minimum token balance to propose.

– *Quorum & Execution:*

  - `quorum(blockNumber)` – Computes required votes for quorum.
  - `state(proposalId)` – Retrieves the state of a given proposal.
  - `proposalNeedsQueuing(proposalId)` – Determines if a proposal requires queuing.

– *Timelock Integration:*

  - `_queueOperations(...)` – Queues a proposal for execution.
  - `_executeOperations(...)` – Executes a successful proposal.
  - `_cancel(...)` – Cancels a pending proposal.
  - `_executor()` – Returns the address authorized to execute proposals.

This ensures seamless governance, integrating proposal creation, voting, and timelock-controlled execution.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Low] Cancellation in GaiaTimelock can't be called from the GaiaGovernor

File(s): `GaiaTimelock.sol`

Description: The cancellation of the proposal is currently done in two places. The first cancellation is reachable by the public function `GaiaGovernor.cancel(...)` and can be called only before the voting period starts. The second proposal cancellation is reachable by the `GaiaTimelock.cancel(...)` and can be called when the voting period has ended, and the proposal is already in the queue. The `GaiaGovernor.cancel(...)` calls `GovernorTimelockControl._cancel(...)` which overrides `Governor._cancel(...)` and adds additional functionality:

```
function _cancel(
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas,
    bytes32 descriptionHash
) internal virtual override returns (uint256) {
    uint256 proposalId = super._cancel(targets, values, calldatas, descriptionHash);
    // @audit: the additional logic that should allow for canceling the proposal after the voting has been done by
    ↪  calling the GaiaTimelock.cancel(....)
    bytes32 timelockId = _timelockIds[proposalId];
    if (timelockId != 0) {
        // cancel
        _timelock.cancel(timelockId);
        // cleanup
        delete _timelockIds[proposalId];
    }
    return proposalId;
}
```

This additional logic, however, is not reachable because it requires the proposal to be in the queue (after voting), but the public function from where it is called `GaiaGovernor.cancel(...)` requires the proposal to be before the voting.

```
function cancel(
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas,
    bytes32 descriptionHash
) public virtual returns (uint256) {
    uint256 proposalId = hashProposal(targets, values, calldatas, descriptionHash);

    // @audit: the cancellation may only be called during the "Pending" state, which is before the voting period
    _validateStateBitmap(proposalId, _encodeStateBitmap(ProposalState.Pending));
    if (_msgSender() != proposalProposer(proposalId)) {
        revert GovernorOnlyProposer(_msgSender());
    }
    return _cancel(targets, values, calldatas, descriptionHash);
}
```

To allow the cancellation of a proposal when it's already queued, the owner may grant the canceller role to another trusted address. This is, however not recommended by OpenZeppelin and stated in the comment in `GaiaGovernorTimelock`:

```
/**
 * WARNING: Setting up the TimelockController to have additional proposers or cancellers besides the governor is very
 * risky, as it grants them the ability to: 1) execute operations as the timelock, and thus possibly performing
 * operations or accessing funds that are expected to only be accessible through a vote, and 2) block governance
 * proposals that have been approved by the voters, effectively executing a Denial of Service attack.
 */
```

Recommendation(s): If the cancellation in the GaiaTimelock is not necessary, there is no need to take any action. However, if it should be reachable, consider adding another public function in GaiaGovernor that would allow calling cancellation on different states than "Pending". Alternatively, the trusted address may be granted the canceller role to call `GaiaTimelock.cancel(...)` directly in emergency situations. Note, however, that this solution brings DOS risk if such an address is compromised, as stated in the OZ comment above.

Status: Acknowledged

Update from the client: Discovered a code segment that does not introduce security vulnerabilities, maintained for potential future architectural considerations.

6.2 [Low] Safeguarding Contract Operations by Disabling Ownership Renouncement

File(s): GaiaToken.sol

Description: The GaiaToken contract inherits from OpenZeppelin's Ownable contract, which provides functionality for the owner to renounce ownership. Once the ownership is renounced, the contract is left without an owner, any functionality restricted to the owner would become permanently inaccessible. For example, GaiaToken extends the Pausable contract, which provides pause and unpause functionalities that can only be executed by the owner.

```
    // @audit this functionality should be disabled
    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }
```

Recommendation(s): To protect the token from accidental renouncement of ownership, it is advisable to explicitly prevent any attempts to renounce ownership. This can be achieved by overriding the renounceOwnership function in the GaiaToken contract. The implementation below ensures the continuity and integrity of pausable operations within the contract:

```
function renounceOwnership() public override onlyOwner {
    revert("Ownership cannot be renounced");
}
```

Status: Fixed

Update from the Nethermind Security: Commit c4afda4.

6.3 [Low] Signers cannot cancel permit signatures before the deadline in GaiaToken

File(s): GaiaToken.sol

Description: GaiaToken permit signature allows a signer to create an EIP-712 signature. However, once the signature is generated, the signer cannot cancel it before the deadline. This limitation arises because the contract is based on OpenZeppelin's ERC20Permit, which does not include an external function for manually increasing the nonce. Instead, it only provides the internal _useNonce function.

Recommendation(s): To enable signers to cancel their signatures, implement an external function that allows them to directly consume their nonce, effectively invalidating the signature. For example:

```
function consumeNonce() external {
    _useNonce(msg.sender);
}
```

Status: Fixed

Update from the Nethermind Security: Commit c4afda4.

6.4 [Low] GaiaToken cannot be paused

File(s): GaiaToken.sol

Description: The GaiaToken contract incorporates an emergency stop mechanism using OpenZeppelin's Pausable contract. The Pausable module provides internal functions to activate and deactivate the paused state, which should only be triggered by an authorized account. However, the GaiaToken contract does not expose external or public functions for utilizing the pausing mechanism.

Recommendation(s): Add external functions that invoke the internal functions to modify the contract's state. Since GaiaToken inherits from the Openzeppelin's Ownable contract, these functions should be restricted to execution exclusively by the contract owner.

Status: Fixed

Update from the Nethermind Security: Commit c4afda4.

6.5 [Low] GaiaToken extends Pausable but does not add pausable functionality

File(s): GaiaToken.sol

Description: A pause feature enables the contract owner or authorized callers to temporarily suspend operations of the contract. The `GaiaTokenEth` contract extends the `Pausable` contract but does not implement the pause mechanism on any function, i.e., `whenNotPaused` and `whenPause` modifiers.

Recommendation(s): Add the operations that should be temporarily halted in the contract in case of an emergency.

Status: Fixed

Update from the client: Fixed

Update from Nethermind Security: In the first round of reaudit, the issue was not fixed. We replied with the following consideration: “the issue is not fixed. To address the described problem, the functions that transfer, mint, or burn tokens should have a `whenNotPaused` modifier. Alternatively, this modifier can be added to the `_udpate(...)` function, which is a common function for all those“. In the second round of reaudit, the issues was fixed in the commit c4afda4.

6.6 [Info] Important deployment information

File(s): GaiaGovernor.sol, GaiaTimelock.sol

Description: The deployment of the GaiaGovernor and GaiaTimelock is difficult since both contracts require each others' addresses. The recommended way of deployment is to first deploy GaiaTimelock and put empty arrays for executors and proposers in the constructor. Then after the deployment of the GaiaGovernor, admin should call the GaiaTimelock to grant the executor and proposer roles to the GaiaGovernor contract.

Recommendation(s): Consider following the proposed scenario during deployment. Additionally read the discussion of this topic here.

Status: Acknowledged

Update from the client: The deployment of GaiaTimelock does not require the address of the GaiaGovernor contract.

Update from Nethermind Security: During the deployment of GaiaTimelock the `proposers` and `executors` are needed to grant proposer, canceller and executor roles (link to the OpenZeppelin implementation). The GaiaGovernor should hold those roles according to the current implementation, therefore it's address is needed during the GaiaTimelock construction.

6.7 [Info] Quorum value may be incorrect because of bridging

File(s): GaiaGovernor.sol

Description: To accept the proposal, a quorum of votes must be reached. The quorum is set by the protocol owner and is calculated as the sum of "for" and "abstain" votes proportional to all the voting power (total minted GAIA tokens). However, with the current setup, the `40%` of all minted tokens are predicted to be bridged to the Base and used there for restaking. The bridged GAIA tokens would be frozen in the bridge contract but would still be accounted for in the calculation of the quorum. That leads to an incorrect quorum. Consider the following example: the owner mints `1000` GAIA tokens. The quorum is set to `10%`, which is `100` tokens. `40%` of those tokens are bridged to Base, leaving `600` tokens on Ethereum. To reach quorum during voting, the proposal must get at least `100` "for/abstain" votes, which constitutes `100/600`, which makes a real quorum at `16.6%` instead of the initial `10%`. Note that the number of bridged tokens may change over time, changing the value of the real quorum.

Recommendation(s): This is a business decision from the client.

Status: Acknowledged

Update from the client: No modifications for now. The quorum threshold can be manually lowered.

6.8 [Info] `GaiaTimelock` could allow anyone to execute proposals

File(s): `GaiaTimelock.sol`

Description: The `GaiaTimelock` contract extends OpenZeppelin's `TimelockController` contract, which enforces a delay between queueing and executing an operation. This contract introduces the flexibility to operate in two modes: **only role** and **open role**. The `onlyRoleOrOpenRole` modifier is used in the `execute` and `executeBatch` functions, as shown below:

```
    function execute(
        address target,
        uint256 value,
        bytes calldata payload,
        bytes32 predecessor,
        bytes32 salt
    ) public payable virtual onlyRoleOrOpenRole(EXECUTOR_ROLE) { ... }

    function executeBatch(
        address[] calldata targets,
        uint256[] calldata values,
        bytes[] calldata payloads,
        bytes32 predecessor,
        bytes32 salt
    ) public payable virtual onlyRoleOrOpenRole(EXECUTOR_ROLE) { ...}
```

Granting `EXECUTOR_ROLE` to the zero address (`address(0)`) enables anyone to execute proposals. The `GaiaTimelock` contract should prohibit setting open roles unless explicitly allowed during deployment. Currently, as presented below, `executors` are set without validation, which could inadvertently permit open roles:

```
    constructor(
        uint256 minDelay,
        address[] memory proposers,
        address[] memory executors,
        address admin
    ) TimelockController(minDelay, proposers, executors, admin) {}
```

Recommendation(s): You can follow the guidelines explained here. Make sure that this role is set before the deployer renounces its admin privileges in favor of `GaiaTimelock` contract itself (for details see TimelockController roles.

Status: Acknowledged

Update from the client: We recognize that role validation is critical and guides future developers to implement proper access controls.

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about GaiaNet documentation

The `GaiaNet` team has provided clarifications through meetings and documentation that outline how the protocol is intended to function, including definitions for each type of node, reward computations, and flow diagrams illustrating interactions between contracts. However, the documentation lacks detailed information, which made it challenging to map some requirements to their implementation in the code. The reward calculations in the code differ from those outlined in the documentation entitled *"Gaianet Token Economy Desgin"*. This specification is essential to identify design flaws and validate the implementation of these requirements from a security perspective.

8 Test Suite Evaluation

8.1 Compilation Output

```
% forge build
[â Š] Compiling...
[â †] Compiling 124 files with 0.8.26
[â ”] Solc 0.8.26 finished in 4.40s
Compiler run successful with warnings:
Warning (2072): Unused local variable.
  --> test/GaiaReward.t.sol:81:13:
   |
81 |             uint256 totalReward,
   |             ^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> test/GaiaReward.t.sol:83:13:
   |
83 |             uint256 dailyReward
   |             ^^^^^^^^^^^^^^^^^^^
```

8.2 Tests Output

```
%forge test
[â Š] Compiling...
No files changed, compilation skipped

Ran 6 tests for test/GDNPaymentReceiver.t.sol:GdnPaymentReceiverTest
[PASS] testFailNonOwnerSetBeneficiary() (gas: 13063)
[PASS] testGetContractBalance() (gas: 60574)
[PASS] testNonBeneficiaryWithdrawShouldRevert() (gas: 13170)
[PASS] testPay() (gas: 60073)
[PASS] testSetBeneficiary() (gas: 16522)
[PASS] testWithdrawUSDT() (gas: 75888)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 4.90ms (2.22ms CPU time)

Ran 6 tests for test/GaiaGovernor.t.sol:GaiaGovernorTest
[PASS] testSetProposalThreshold() (gas: 24408)
[PASS] testSetVotingDelay() (gas: 24817)
[PASS] testSetVotingDelayValidRange() (gas: 27797)
[PASS] testSetVotingPeriod() (gas: 27405)
[PASS] testTokenVotingPower() (gas: 89751)
[PASS] testUpdateQuorumNumerator() (gas: 31613)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 6.36ms (1.53ms CPU time)

Ran 13 tests for test/GaiaReward.t.sol:GaiaRewardTest
[PASS] testBatchSizeLimit() (gas: 184287)
[PASS] testCombinedRewardDistribution() (gas: 199901)
[PASS] testDailyRewardCalculation() (gas: 19651)
[PASS] testDistributionInputValidation() (gas: 138289)
[PASS] testDomainRewardDistribution() (gas: 189149)
[PASS] testEmergencyReset() (gas: 95477)
[PASS] testMultiRoundDistribution() (gas: 243632)
[PASS] testNodeRewardDistribution() (gas: 188629)
[PASS] testPermissions() (gas: 27445)
[PASS] testRewardClaim() (gas: 246340)
[PASS] testRoundManagement() (gas: 125282)
[PASS] testSpecificNodeRewardDistribution() (gas: 162229)
[PASS] testStakerRewardDistribution() (gas: 208422)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 6.68ms (1.94ms CPU time)
```

```
Ran 7 tests for test/GdnReferralRewardClaimerTest.t.sol:GdnReferralRewardClaimerTest
[PASS] testClaimReward() (gas: 103216)
[PASS] testClaimRewardWithExpiredSignature() (gas: 47895)
[PASS] testClaimRewardWithInvalidSignature() (gas: 45178)
[PASS] testSetSigner() (gas: 19546)
[PASS] testSetSignerNotOwner() (gas: 13179)
[PASS] testWithdrawUSDTInsufficientBalance() (gas: 36301)
[PASS] testWithdrawUSDTSuccess() (gas: 48677)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 6.85ms (3.15ms CPU time)

Ran 6 tests for test/GaiaCredit.t.sol:GaiaCreditTest
[PASS] testConsume() (gas: 141372)
[PASS] testDeposit() (gas: 131862)
[PASS] testFailUnauthorizedDepositor() (gas: 984540)
[PASS] testRedeem() (gas: 74043)
[PASS] testSetMaxOverdraft() (gas: 16436)
[PASS] testWithdrawUSDT() (gas: 133419)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 7.89ms (7.58ms CPU time)

Ran 33 tests for test/GDN.t.sol:GdnContractsTest
[PASS] testBeneficiaryWithdrawSucceeds() (gas: 223742)
[PASS] testEncodeDecode() (gas: 46566)
[PASS] testGetContractBalance() (gas: 51320)
[PASS] testGetLabelInfo() (gas: 235346)
[PASS] testIsLabelAvailable() (gas: 237263)
[PASS] testNonBeneficiaryWithdrawFails() (gas: 242659)
[PASS] testNonOwnerSetBeneficiary() (gas: 18807)
[PASS] testNonOwnerSetRegistrar() (gas: 13357)
[PASS] testNonOwnerSetRegistrarInService() (gas: 13234)
[PASS] testNonOwnerSetRegistrationService() (gas: 13193)
[PASS] testNonOwnerSetRegistryInRegistrar() (gas: 13213)
[PASS] testNonOwnerSetRegistryInService() (gas: 13192)
[PASS] testNonOwnerSetSigner() (gas: 13170)
[PASS] testNonOwnerTransferNFT() (gas: 273348)
[PASS] testNonOwnerWithdraw() (gas: 13162)
[PASS] testNonRegistrarSetLabel() (gas: 14144)
[PASS] testNonRegistrationServiceMint() (gas: 14093)
[PASS] testNonRegistrationServiceRenew() (gas: 13762)
[PASS] testOwnerSetBeneficiary() (gas: 21531)
[PASS] testOwnerWithdraw() (gas: 220554)
[PASS] testReRegisterExpiredLabel() (gas: 387366)
[PASS] testReRegisterExpiredLabelAsNew() (gas: 366322)
[PASS] testReRegisterExpiredLabelByOriginalOwner() (gas: 328665)
[PASS] testRegisterExistingLabel() (gas: 245873)
[PASS] testRegisterLabelWithZeroAmountPaid() (gas: 213469)
[PASS] testRegisterNewLabel() (gas: 245809)
[PASS] testRenewLabel() (gas: 340540)
[PASS] testSetBaseTokenURI() (gas: 29534)
[PASS] testSetTokenURIExtension() (gas: 28810)
[PASS] testTokenURI() (gas: 108172)
[PASS] testTransferExpiredLabel() (gas: 263536)
[PASS] testTransferLabel() (gas: 285946)
[PASS] testWithdrawUSDTInsufficientBalance() (gas: 52940)
Suite result: ok. 33 passed; 0 failed; 0 skipped; finished in 7.92ms (24.22ms CPU time)

Ran 6 test suites in 121.79ms (40.60ms CPU time): 71 tests passed, 0 failed, 0 skipped (71 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.