Security Review Report
NM-0731 - idOS Network - Node Staking

NETHERMIND
SECURITY

(November 27, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for idOS Node Staking contracts.

The **IDOS Protocol** is a decentralized staking framework built on Arbitrum One. The system enables token holders to participate in network security by delegating their **IDOSToken** assets to verified node operators. In return for securing the network, participants earn staking rewards distributed through an epoch-based algorithmic model, with each epoch lasting 1 day. The protocol is modular and consists of a core staking contract, a custom ERC20 token implementation, and a vesting utility for linear token distribution. The system also implements a slashing mechanism that the off-chain governance can use in order to punish nodes that misbehave.

The audit comprises 330 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 6 points of attention. where one is classified as `Critical`, one is classified as `High`, and four are classified as `Low` severity. The issues are summarized in Fig. 1.

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

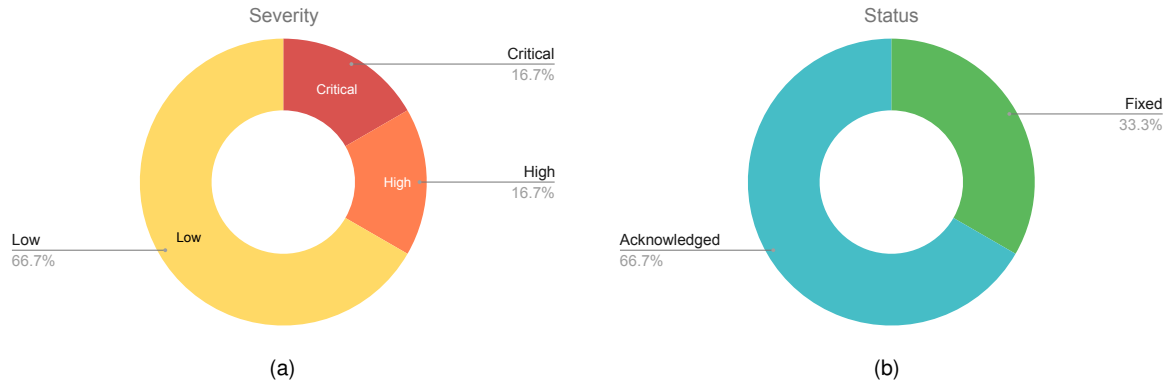

Fig. 1: Distribution of issues: Critical (1), **High** (1), **Medium** (0), **Low** (4), **Undetermined** (0), **Informational** (0), **Best Practices** (0). **Distribution of status: Fixed** (2), **Acknowledged** (4), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	November 24, 2025
Final Report	November 27, 2025
Initial Commit	e2989c5
Final Commit	c0d3507
Documentation Assessment	Low
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	IDOSVesting.sol	24	9	37.5%	4	37
2	IDOSNodeStaking.sol	282	7	2.5%	72	361
3	IDOSToken.sol	24	2	8.3%	6	32
	Total	**330**	**18**	**5.5%**	**82**	**430**

3 Summary of Issues

	Finding	Severity	Update
1	The `stake` function doesn't check if the node is `allowListed`	Critical	Fixed
2	The `withdrawableReward` function uses incorrect reward values for epochs	High	Fixed
3	Unbounded loop in `withdrawUnstaked(...)` results in potential Denial of Service	Low	Acknowledged
4	Unbounded loop in `withdrawableReward(...)` leads to Denial of Service for new and inactive stakers	Low	Acknowledged
5	Users can evade slashing penalties by front-running the `slash(...)` transaction	Low	Acknowledged
6	Users can manipulate reward distribution by staking at the end of an epoch	Low	Acknowledged

4 System Overview

The IDOS Protocol is a decentralized staking framework built on Arbitrum One. The system enables token holders to participate in network security by delegating their `IDOSToken` assets to verified node operators. In return for securing the network, participants earn staking rewards distributed through an epoch-based algorithmic model. The protocol is modular, consisting of a core staking contract, a custom ERC20 token implementation, and a vesting utility for linear token distribution.

4.1 Staking and Delegation

The entry point for user participation is the `IDOSNodeStaking` contract. To participate, users must possess `IDOSToken` assets and delegate them to a specific node operator.

- **Delegation Mechanism**: Users invoke the `stake(...)` function, specifying the amount of tokens and the target node address.
- **State Management**: Upon a successful stake, the protocol updates the global state to reflect the increased Total Value Locked (TVL) for the current epoch. It simultaneously updates the individual mapping of the user's stake against the chosen node.
- **Token Custody**: Staked tokens are transferred from the user's wallet to the `IDOSNodeStaking` contract, where they are held in custody until the user initiates an unstaking procedure.

4.2 Epoch-Based Reward Distribution

The protocol utilizes a discrete time model divided into epochs, with each epoch defined by the constant `EPOCH_LENGTH` (set to 1 day). Rewards are calculated based on the state of the system and the reward rate active during each specific epoch.

- **Reward Accumulators**: Instead of updating every user's balance at the start of a new epoch, the contract maintains global accumulators (`rewardAcc`, `userStakeAcc`) to track the total rewards per staked token over time.
- **Checkpoints**: When a user interacts with the contract (via staking, unstaking, or claiming), the system triggers the `createEpochCheckpoint(...)` function. This function synchronizes the user's local state with the global accumulators, iterating through all epochs that have elapsed since the user's last interaction.
- **Epoch Rewards**: The protocol administration defines the reward amount distributed per epoch. Updates made via `setEpochReward` are persistent; a new reward value applies to the current epoch and remains active for all subsequent epochs until explicitly changed. During calculation, the system applies the specific reward rate that was valid during each iterated epoch.

4.3 Unstaking and Withdrawals

To ensure network stability and prevent capital flight, the protocol enforces a mandatory unbonding period for all staked assets.

- **Initiating Unstake**: Users call `unstake(...)` to request the withdrawal of their assets. This action moves the tokens from the "Active Stake" state to a "Pending Withdrawal" state. The protocol records the timestamp of this request.
- **Unbonding Period**: The `UNSTAKE_DELAY` is configured to 14 days. During this window, the assets are locked and do not earn rewards.
- **Finalizing Withdrawal**: Once the delay period has elapsed, users can execute `withdrawUnstaked()`. This function iterates through the user's unstake requests, aggregates all mature requests, and transfers the principal `IDOSToken` amount back to the user's wallet.
- **Claiming Rewards**: Distinct from principal withdrawals, accrued yield can be claimed at any time via the `withdrawReward()` function without affecting the principal stake.

4.4 Node Management and Slashing

The IDOS Protocol relies on a permissioned set of node operators to maintain network integrity. The management of these nodes is handled by the protocol owner.

- **Node Allowlisting**: The system is designed to support a compact set of verified nodes (approximately 20), ensuring efficient iteration during state updates.
- **Slashing Mechanism**: If a node acts maliciously, the owner can invoke the `slash(...)` function. This action immediately freezes the node, preventing further stakes.
- **Impact on Stakers**: When a node is slashed, the stakes delegated to that node are removed from the "Active Stake" calculation for the users. These funds are effectively confiscated from the user's view and become withdrawable by the protocol owner via `withdrawSlashedStakes()`, allowing for off-chain redistribution or burning depending on governance decisions.

4.5 Vesting Architecture

To support long-term alignment of incentives, the protocol includes an `IDOSVesting` contract. This contract extends the standard OpenZeppelin `VestingWallet` with "Cliff" functionality. It allows the protocol to allocate tokens to beneficiaries (such as team members, early investors, or the protocol treasury) that unlock linearly over a specified duration, but only after a defined `cliffSeconds` threshold has passed.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security 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 the finding is to be uncovered and exploited by an attacker. 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 Security 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 pass to the client formally;

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 [Critical] The `stake` function doesn't check if the node is `allowListed`

File(s): contracts/IDOSNodeStaking.sol

Description: The `IDOSNodeStaking.stake(...)` function doesn't check if the input parameter `node` is part of the `allowlistedNodes` set.

```
function stake(address user, address node, uint256 amount) external nonReentrant whenNotPaused {
    //@audit--issue lack of checks that the node is actually allowlisted
    require(node != address(0), ZeroAddressNode());
    require(!slashedNodes.contains(node), NodeIsSlashed(node));
    require(amount > 0, AmountNotPositive(amount));
    require(block.timestamp >= startTime, NotStarted());

    if (user == address(0)) user = msg.sender;
    //..
    //..
}

```

This allows users to pick any arbitrary address as the `node` to stake with. A user can even pick their own address as the `node`. This would allow a `node` to be both a staker and node at the same time. The problem is that these stakers will accrue rewards, just like the ones who staked on the `allowlistedNodes` would and this enables malicious users to steal the staking rewards from honest stakers and nodes.

As a side note, the `unstake(...)` and `slash(...)` functions don't implement this check either.

Recommendation(s): Consider adding a check that the `node` input parameter is part of the `allowlistedNodes` list.

Status: Fixed

Update from the client: Fixed at commit

6.2 [High] The `withdrawableReward` function uses incorrect reward values for epochs

File(s): `contracts/IDOSNodeStaking.sol`

Description: The `withdrawableReward(...)` function in the `IDOSNodeStaking` contract calculates the pending rewards for a user by iterating through the epochs that have passed since the user's last checkpoint. The protocol starts from a default epoch reward value (stored at key `0`) and allows the owner to change the epoch reward via `setEpochReward`. Each call to `setEpochReward` at epoch `X` is intended to define a new reward value that applies from epoch `X` **onward** until changed again (i.e. rewards are _sticky_ across epochs).

The issue lies in how the `epochReward` variable is initialized relative to the user's checkpoint. At the beginning of `withdrawableReward(...)`, the `epochReward` variable is always set to the default reward at index `0`, regardless of any changes that may have occurred before the user's last checkpoint. The function then iterates from `checkpoint.epoch` up to `currentEpoch()`, and only updates `epochReward` if there is an entry in `epochRewardChanges` at the _exact_ epoch `i`:

```
function withdrawableReward(address user)
    public
    view
    returns (uint256 withdrawableAmount, uint256 rewardAcc, uint256 userStakeAcc, uint256 totalStakeAcc)
{
    // ...
    //@audit rewards are set to the default value
    uint256 epochReward = epochRewardChanges.get(0);

    for (uint48 i = checkpoint.epoch; i < currentEpoch(); i++) {
        //@audit if rewards were not updated exactly at epoch `i` the protocol
        //uses the default value, which may be incorrect
        (bool exists, uint256 rewardAtEpoch) = epochRewardChanges.tryGet(i);
        if (exists) epochReward = rewardAtEpoch;

        // ...
        if (totalStakeAcc == 0) continue;
        rewardAcc += (userStakeAcc * epochReward) / totalStakeAcc;
    }

//..
}
```

Whenever the epoch reward has been changed at some epoch `X` and a user's checkpoint is at a later epoch `checkpoint epoch > X`, the rewards for epochs >= user's checkpoint are computed using the default reward instead of the last changed reward. This will lead to incorrect rewards calculations for stakers.

Recommendation(s): Consider revisiting the loop logic to ensure correct rewards calculations.

Status: Fixed

Update from the client: Fixed at commit

6.3 [Low] Unbounded loop in `withdrawUnstaked(...)` results in potential Denial of Service

File(s): contracts/IDOSNodeStaking.sol

Description: The `unstake(...)` function allows users to initiate the withdrawal process by appending an `Unstake` struct to the `unstakesByUser` array. After the `UNSTAKE_DELAY` period, users call the `withdrawUnstaked(...)` function to claim their tokens.

However, the `withdrawUnstaked(...)` function iterates over the entire `unstakesByUser` array in a single transaction to find eligible withdrawals.

```
function withdrawUnstaked()
    external nonReentrant whenNotPaused
    returns (uint256 withdrawableAmount)
{
    withdrawableAmount;
    // @audit The loop iterates over the entire array which can grow indefinitely
    for (uint i; i < unstakesByUser[msg.sender].length; i++)
        if (unstakesByUser[msg.sender][i].timestamp < uint48(block.timestamp) - UNSTAKE_DELAY) {
            withdrawableAmount += unstakesByUser[msg.sender][i].amount;
            // @audit-issue 'delete' zeroes the element but does not reduce array length
            delete unstakesByUser[msg.sender][i];
        }
    // ...
}
```

This logic presents two significant risks that can cause the transaction to exceed the block gas limit, leading to a Denial of Service (DoS) where funds become permanently locked:

1. **Inefficient Array Management**: The code uses `delete` to clear processed unstakes. In Solidity, `delete` only resets the value to default (zero) but does not remove the index or decrease the array length. Consequently, the array grows indefinitely, forcing the loop to process an increasing number of "ghost" entries from past withdrawals;
2. **Accumulation of Pending Unstakes**: If a user performs many unstake operations without calling `withdrawUnstaked(...)` for a long period, the array will contain a large number of valid, pending entries. Attempting to process all of them in a single loop may consume more gas than the block limit allows;

Recommendation(s): Consider refactoring the withdrawal logic to prevent unbounded iteration and support batch processing. Two potential solutions include:

1. **Start Index and Batching**: Implement a pointer or start index variable to track the head of the queue (the first unprocessed withdrawal). Combine this with a parameter that allows the user to specify the number of items to process in one transaction;
2. **Swap-and-Pop and Batching**: Replace the `delete` operation with a "swap and pop" mechanism. This involves moving the last element of the array to the position of the withdrawn element and then popping the array to decrease its length. This should also be combined with a user-provided parameter to limit the number of withdrawals processed per transaction;

Status: Acknowledged

Update from the client: We ran the following test: after 700 noisy epochs (with 100 other users staking, unstaking, etc), we had a user stake every day and unstake every other day for another 700 quiet epochs; last unstake cost 200k gas.

6.4 [Low] Unbounded loop in `withdrawableReward(...)` leads to Denial of Service for new and inactive stakers

File(s): contracts/IDOSNodeStaking.sol

Description: The `stake(...)`, `unstake(...)` and `withdrawReward(...)` functions utilize the `createEpochCheckpoint(...)` function to update the user's state. This internal logic relies on `withdrawableReward(...)` to calculate the pending rewards accumulated since the user's last interaction.

However, the `withdrawableReward(...)` function calculates rewards by iterating from the user's last recorded epoch (`checkpoint.epoch`) up to the `currentEpoch()`. For a new user who has never interacted with the contract, the default value of `checkpoint.epoch` is `0`.

```
function withdrawableReward(address user)
    public view
    returns (uint256 withdrawableAmount, uint256 rewardAcc, uint256 userStakeAcc, uint256 totalStakeAcc)
{
    EpochCheckpoint memory checkpoint = epochCheckpointByUser[user];
    // ...
    // @audit If the user is new, checkpoint.epoch is 0.
    // @audit If the protocol has been running for a long time, this loop becomes extremely large.
    for (uint48 i = checkpoint.epoch; i < currentEpoch(); i++) {
        // ...
    }

    // ...
}
```

This creates a Denial of Service (DoS) vector in two scenarios:

1. **New Stakers**: If the protocol has been active for a long time (a large number of epochs have passed), a new user trying to `stake(...)` will trigger the loop from epoch 0 to the current epoch. If the gas cost of this iteration exceeds the block gas limit, the user effectively cannot stake;
2. **Inactive Stakers**: Existing users who have not interacted with the contract for a long period will face the same issue when trying to `unstake(...)` or claim rewards;

Recommendation(s): Consider implementing the following changes to mitigate the DoS risk:

1. **Pagination**: Update the logic to allow processing epochs in batches. This allows users to update their checkpoints incrementally across multiple transactions rather than forcing the calculation of all missing epochs in a single transaction;
2. **Optimize for New Users**: Check if a user is a new staker (e.g., has a stake of 0 and no history). If so, set their `checkpoint.epoch` directly to `currentEpoch()` to avoid iterating through history where they had no stake;
3. **Distinguish Epoch 0**: To safely distinguish between a "new user" (default 0) and a user who genuinely staked at epoch 0, ensure the first valid epoch of the protocol is `1` (or higher) by adjusting the `startTime` in the constructor;

Note regarding economic viability: While pagination solves the hard DoS limits, if the gas cost to process a single epoch iteration exceeds the rewards earned in that epoch, the function becomes economically unviable to execute. However, given that the contract is deployed on Arbitrum One which is expected to have low gas fees, this specific economic edge case is considered unlikely to happen.

Status: Acknowledged

Update from the client: If a user stakes after 1400 epochs, the transaction still has enough gas to go through. Furthermore, the contract will not be used for more than a couple of months before being replaced, so it will be replaced before this issue can occur.

6.5 [Low] Users can evade slashing penalties by front-running the `slash(...)` transaction

File(s): `contracts/IDOSNodeStaking.sol`

Description: The `IDOSNodeStaking` contract allows users to delegate their tokens to nodes via the `stake(...)` function. If a node acts maliciously, the contract owner can call the `slash(...)` function to penalize the node. This action is intended to freeze the stakes associated with that node, preventing withdrawals. Users can normally retrieve their funds by calling `unstake(...)`, waiting for the `UNSTAKE_DELAY`, and then calling `withdrawUnstaked(...)`.

The vulnerability lies in the fact that `unstake(...)` immediately reduces the staked balance associated with the node in the `stakeByNode` and `stakeByNodeByUser` mappings. If a user anticipates that a node is about to be slashed (e.g., through off-chain monitoring or governance announcements), they can call `unstake(...)` before the `slash(...)` transaction is executed.

Once the user calls `unstake(...)`, the funds are moved to a pending state in the `unstakesByUser` array, effectively dissociating them from the node. When `slash(...)` is subsequently executed, it only locks the funds currently staked on the node. The `withdrawUnstaked(...)` function does not verify if the node associated with the pending withdrawal was slashed. Consequently, the user can withdraw their funds after the delay, successfully bypassing the slashing penalty.

Although the contract is deployed on Arbitrum One, which mitigates standard mempool front-running, this issue remains relevant if the slashing decision is predictable or communicated off-chain prior to on-chain execution.

```
function unstake(address node, uint256 amount)
    external nonReentrant whenNotPaused
{
    // ...
    // @audit Checks if node is currently slashed, but doesn't prevent future slashing evasion
    require(!slashedNodes.contains(node), NodeIsSlashed(node));

    // ...
    // @audit-issue User stake is immediately decremented from the node mapping
    stakeByNodeByUser[msg.sender][node] -= amount;

    // ...
    // @audit-issue Node total stake is immediately reduced
    uint256 newNodeStake = getNodeStake(node) - amount;
    if (newNodeStake > 0) {
        stakeByNode.set(node, newNodeStake);
    } else {
        stakeByNode.remove(node);
    }

    // ...
    // @audit The pending unstake does not track which node these funds came from
    unstakesByUser[msg.sender].push(Unstake(amount, uint48(block.timestamp)));

    // ...
}
```

Recommendation(s): Consider updating the `Unstake` struct to store the address of the node being unstaked. In the `withdrawUnstaked(...)` function, introduce logic to verify if the corresponding node has been slashed.

To ensure fairness, distinguish between long-standing unstake requests and those made immediately prior to a slashing event. Consider checking if the node was slashed within a specific window relative to the unstake timestamp. If the slash happened shortly after the unstake, the withdrawal should be blocked or penalized.

Crucially, if a user's withdrawal is blocked due to slashing, ensure the system correctly accounts for these funds so they are recoverable by the protocol. Since `unstake(...)` reduces the `stakeByNode` mapping, these funds would not normally be captured by the `withdrawSlashedStakes(...)` function (which iterates over current node stakes). The logic must essentially revert the accounting reductions made during `unstake(...)` or explicitly route these blocked funds to the slashed pool to ensure they can be withdrawn by the owner.

Status: Acknowledged

Update from the client: This applies only to "off-chain front-running" since Arbitrum One has no public mempool. And validators won't risk slashing by announcing slashable offenses publicly. We don't really expect slashing with the small and trusted validator set we'll have for the foreseeable future (note again this contract is unlikely to live for more than a few months).

6.6 [Low] Users can manipulate reward distribution by staking at the end of an epoch

File(s): contracts/IDOSNodeStaking.sol

Description: The `IDOSNodeStaking` contract incentivizes users to stake tokens delegated to nodes by distributing rewards over fixed time periods called epochs. The duration of an epoch is defined by `EPOCH_LENGTH`, which is set to 1 day. Users can stake tokens using the `stake(...)` function and claim their accumulated rewards via `withdrawReward(...)`.

The issue arises from the way rewards are calculated relative to the time of staking. When a user calls `stake(...)`, their stake is recorded for the current epoch. Subsequently, the `withdrawableReward(...)` function calculates rewards by iterating through completed epochs. It uses the total stake recorded for a specific epoch to determine the user's share of the rewards. However, the calculation does not account for the duration the tokens were actually staked within that epoch.

This allows a malicious user to deposit a large amount of tokens just before the epoch ends (e.g., in the last second). As soon as the next epoch begins, the protocol considers their stake to have been active for the entire duration of the previous epoch. Consequently, the attacker receives the full reward for that epoch despite their funds being exposed to the protocol (and potential slashing risks) for only a brief moment. This behavior dilutes the rewards of honest stakers who risk their capital for the full duration.

The `stake(...)` function updates the mapping for the current epoch immediately:

```
function stake(address user, address node, uint256 amount)
    external nonReentrant whenNotPaused
{
    // ...
    stakeByNodeByUser[user][node] += amount;
    // @audit The stake is added to the current epoch's balance immediately
    stakeByUserByEpoch[currentEpoch()][user] += amount;
    stakedByEpoch[currentEpoch()] += amount;
    // ...
}
```

The `withdrawableReward(...)` function calculates the reward based on that balance, treating it as valid for the whole epoch:

```
function withdrawableReward(address user)
    public view
    returns (uint256 withdrawableAmount, uint256 rewardAcc, uint256 userStakeAcc, uint256 totalStakeAcc)
{
    // ...
    for (uint48 i = checkpoint.epoch; i < currentEpoch(); i++) {
        // ...
        // @audit The user's stake is added to the accumulator for the entire epoch `i`
        userStakeAcc += stakeByUserByEpoch[i][user];
        userStakeAcc -= unstakeByUserByEpoch[i][user];

        totalStakeAcc += stakedByEpoch[i];
        // ...
        // @audit Rewards are distributed based on the final stake of the epoch
        rewardAcc += (userStakeAcc * epochReward) / totalStakeAcc;
    }
    // ...
}
```

Recommendation(s): Consider modifying the staking logic so that deposits made during the current epoch only become eligible for rewards starting from the subsequent epoch. This ensures that a user's funds must be staked for at least one full epoch cycle before earning rewards, aligning the reward incentive with the risk exposure.

Status: Acknowledged

Update from the client: This was a deliberate trade-off. We eschewed the complexity of implementing a more complex AMM/staking pattern (e.g. Synthetix, Uniswap) by using short 1-day epochs combined with a 14-day unbonding period.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. 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 they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about idOS's documentation

The **idOS** team provided a walkthrough of the in-scope contracts during the kickoff call. The team addressed the questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```
> npx hardhat test
Running Solidity tests

  IDOSNodeStaking
    Pausing
      When not paused
         Can be paused only by owner
      When paused
         Can be unpaused by owner
         Can't allowNode
         Can't disallowNode
         Can't stake
         Can't unstake
         Can't withdrawUnstaked
         Can't withdrawSlashedStakes
         Can't withdrawReward
    Allowlisting
       Node can be allowed only by owner
       Node can be disallowed only by owner
       Emits events
    Staking
      Before starting
         Can't stake yet
      After starting
         Epochs last 1 day (49ms)
         Can't stake against zero address
         Can't stake against slashed node
         Can only stake positive amounts
         Emits events
         Works (115ms)
    Unstaking
      Before starting
         Can't unstake yet
      After starting
         Can't unstake from zero address
         Can't unstake from slashed node
         Can only unstake positive amounts
         Can only unstake up to staked amount
         Emits events
         Works
        Withdrawal
           Can't withdraw before delay
           Emits events
           Works (55ms)
    Slashing
       Unknown nodes can't be slashed (39ms)
       Slashed nodes can't be slashed
       Known nodes can be slashed only by owner
      Withdrawing slashed stakes
         Can be done only by owner
         Emits events
         Works
    Rewards
       Count only past epochs
       Ignore slashed stakes
       changes value according to the epoch reward (40ms)
       changes value according to the epoch reward and keeps track of past epoch rewards
       Works I (39ms)
       Works II (135ms)
       Works III (148ms)
```

```
IDOSToken
   Should premint 1B tokens
   Should not allow minting
   Should allow burning
   Should allow pausing and unpausing by owner

IDOSVesting
   Should work without cliff (54ms)
   Should work with cliff (58ms)
48 passing (2s)
```

8.2 Automated Tools

8.2.1 AuditAgent

The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

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 conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to 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.

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.