Security Review Report
NM-0338 - Summon Staker

(October 21, 2024)

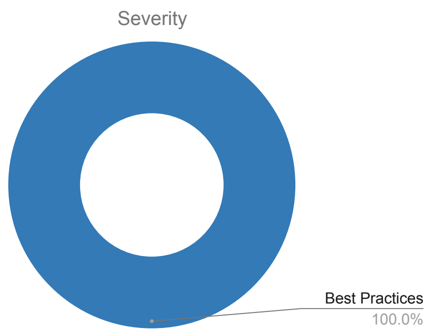
Contents

1 Executive Summary

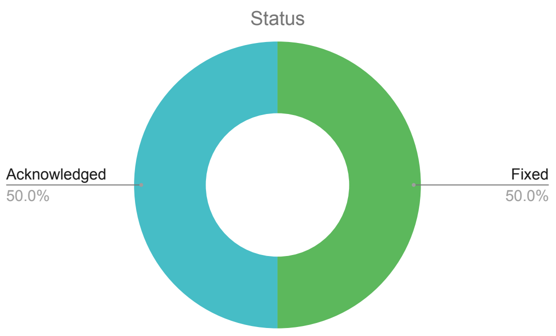
This document outlines the security review conducted by Nethermind Security for the Summon staker contract. The audit covers a permissionless staking protocol that allows deposits for native, `ERC20`, `ERC721`, and `ERC1155` tokens.

The audited code comprises 324 lines of code written in the Solidity language. The Summon team has provided comprehensive documentation, including design documents and inline comments to explain the protocol's behavior. Additionally, the Nethermind and Summon teams have communicated to clarify all other questions regarding protocol design.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contract. Along with this document, we report 2 points classified as `best practices`. The issues are summarized in Fig. 1.



(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** (0), **Undetermined** (0), **Informational** (0), **Best Practices** (2). **(b) Distribution of status: Fixed** (1), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	October 21, 2024
Final Report	October 25, 2024
Methods	Manual Review, Automated analysis
Repository	`G7DAO/protocol/web3/contracts/staking/Staker.sol`
Commit Hash	`bef2650f8ef692586e95f69d5b46260e89c1c3ca`
Final Commit Hash	`https://github.com/G7DAO/protocol/pull/150/files`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	Staker.sol	324	138	42.6%	78	540
	Total	**324**	**138**	**42.6%**	**78**	**540**

3 Summary of Issues

	Finding	Severity	Update
1	Pool administrators can lock tokens using `cooldown` and `lockup` durations	Best Practices	Acknowledged
2	Staking functions do not follow the Checks-Effects-Interactions pattern	Best Practices	Fixed

4 System Overview

The Summon protocol implements the `Staker` contract, which is a permissionless staking contract where any user can create a staking pool to be used by others. The staking contract supports deposits of native, `ERC20`, `ERC721`, and `ERC1155` assets. Any user can create a staking pool and set configuration values for the token lock durations, asset type, and position transferability. Once a pool has been created any user is able to deposit into this pool. Each stake position is represented as an `ERC721` asset which may be transferred, depending on the pool transferability options set by the pool creator. Pool creators are able to change the configuration after it has been created, and ownership of a pool can be transferred to the zero address to prevent future config changes. Asset lock duration is broken down into two sections; `lockup` and `cooldown`. When a stake begins, the lockup period starts to count down. Once the lockup time has passed, then the user must initiate an unstake, where the cooldown period will begin. Once the cooldown period has passed the user can finish the unstake process and receive their assets.

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 [Best Practice] Pool administrators can lock tokens using cooldown and lockup durations

File(s): web3/contracts/staking/Staker.sol

Description: Every pool has an administrator who can modify some aspects of the pool through the function updatePoolConfiguration(...). Specifically, the administrator can change the transferability, lockupSeconds, and cooldownSeconds. For the lockup and cooldown durations, there are no upper limits on how long a user must wait, meaning it is possible for an administrator to effectively lock user tokens by setting the duration to an extremely long duration. The code with audit comments is reproduced below.

```
function updatePoolConfiguration(...) external {
    StakingPool storage pool = Pools[poolID];

    ...

    ///////////////////////////////////////////////////////////////////////////
    // @audit: missing input validation for lockupSeconds
    ///////////////////////////////////////////////////////////////////////////
    if (changeLockup) {
        pool.lockupSeconds = lockupSeconds;
    }

    ///////////////////////////////////////////////////////////////////////////
    // @audit: missing input validation for cooldownSeconds
    ///////////////////////////////////////////////////////////////////////////
    if (changeCooldown) {
        pool.cooldownSeconds = cooldownSeconds;
    }
    emit StakingPoolConfigured(
        poolID,
        pool.administrator,
        pool.transferable,
        pool.lockupSeconds,
        pool.cooldownSeconds
    );
}
```

The values of lockupSeconds and cooldownSeconds are also set during the pool's creation without any validation, as reproduced below.

```
function createPool(..., uint256 lockupSeconds, uint256 cooldownSeconds, address administrator) external {

    ...

    ///////////////////////////////////////////////////////////////////////////
    // @audit: missing input validation for cooldownSeconds and lockupSeconds
    ///////////////////////////////////////////////////////////////////////////
    Pools[TotalPools] = StakingPool({
        administrator: administrator,
        tokenType: tokenType,
        tokenAddress: tokenAddress,
        tokenID: tokenID,
        transferable: transferable,
        lockupSeconds: lockupSeconds,
        cooldownSeconds: cooldownSeconds
    });

    ...
}
```

Also, if the `cooldownSeconds` is set to a value such that when added with a position's stake timestamp, it exceeds `2^256-1`, then it is possible to overflow when calling `initiateUnstake`, making it impossible to unstake tokens. The function is reproduced below.

```
function initiateUnstake(uint256 positionTokenID) external nonReentrant {
    ...

    ///////////////////////////////////////////////////////////////////////////
    // @audit: in case of overflow, the transaction reverts.
    //         Unstake will be impossible.
    ///////////////////////////////////////////////////////////////////////////
    if (block.timestamp < position.stakeTimestamp + pool.lockupSeconds) {
        revert LockupNotExpired(position.stakeTimestamp + pool.lockupSeconds);
    }

    ...
}
```

Recommendation(s): Consider adding input validation when setting the `lockupSeconds` and `cooldownSeconds` to prevent these values from being unreasonably high. This validation should be added to both the `constructor` and `updatePoolConfigurations`.

Status: Acknowledged

Update from client: We don't think this is a protocol-level concern. Applications which use the 'Staker' may want different types of protections on these values. For example, we are planning to have game contracts which default to max lockup but modify it atomically for players based on game state.

6.2 [Best Practice] Staking functions do not follow the Checks-Effects-Interactions pattern

File(s): `web3/contracts/staking/Staker.sol`

Description: The stake functions in the `Staker` contract interact with external contracts without following the Checks-Effects-Interactions (CEI) pattern. The Checks-Effects-Interactions Pattern organizes function logic into three steps: a) Checks (validate initial conditions and prerequisites); b) Effects (update the contract's state; c) Interactions (make external calls or transfer `ETH`).

This pattern mitigates the risk of reentrancy attacks by ensuring that state changes occur before any external interactions. Even if a malicious contract calls back into your function, the state has already been updated, preventing unintended behavior.

The functions that require changes are listed below.

- `stakeERC20(...);`
- `stakeERC721(...);`
- `stakeERC1155(...);`
- `stakeNative(...);`

Those functions use the `nonReentrant` modifier. This modifier prevents a function from being called recursively. It locks the function during its execution to avoid reentrancy by ensuring that the same function cannot be entered again before the current execution completes. However, this only protects against direct reentrancy from the same contract, and it doesn't address all security risks related to external interactions.

The Checks-Effects-Interactions pattern helps prevent (a) state inconsistencies in case of failures during external calls and (b) hard-to-trace intermediate states in more complex logic.

Recommendation(s): Move the external interactions to the last line of the function.

Status: Fixed

Update from the client: Fixed in https://github.com/G7DAO/protocol/pull/150/files.

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. It enables effective communication between developers, testers, users, and other stakeholders. It helps 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 it in the future.

Remarks about the Documentation

The client has provided documentation about their protocol through their github documentation, which includes the execution flow of the Staker contract. Additionally, the client was available to address any questions or concerns from the Nethermind Security team.

8 Test Suite Evaluation

8.1 Compilation Output

```
% npx hardhat compile
(node:41335) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative
 ↪ instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
Generating typings for: 50 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 130 typings!
Compiled 49 Solidity files successfully (evm target: paris).
```

8.2 Tests Output

8.2.1 Staker tests

```
% npx hardhat test

  Staker
     STAKER-1: Anybody should be able to deploy a Staker contract.
     STAKER-2: The Staker implements ERC721
     STAKER-3: Token types
     STAKER-4: Any account should be able to create a staking pool for native tokens.
     STAKER-5: Any account should be able to create a staking pool for ERC20 tokens.
     STAKER-6: Any account should be able to create a staking pool for ERC721 tokens.
     STAKER-7: Any account should be able to create a staking pool for ERC1155 tokens.
     STAKER-8: Staking pool IDs should start at 0 and increase sequentially, with correct token types.
     STAKER-9: It should not be possible to create a staking pool for a token of an unknown type.
     STAKER-10: It should not be possible to create native token staking pools with non-zero token address or token ID.
     STAKER-11: It should not be possible to create ERC20 token staking pools with zero token address or non-zero token
     ↪  ID.
     STAKER-12: It should not be possible to create ERC721 token staking pools with zero token address or non-zero token
     ↪  ID.
     STAKER-13: It should not be possible to create ERC1155 token staking pools with zero token address.
     STAKER-14: It should be possible to create ERC1155 token staking pools in which the token ID is zero.
     STAKER-15: An administrator should be able to modify any subset of the configuration parameters on a pool in a
     ↪  single transaction
     STAKER-16: A non-administrator (of any pool) should not be able to change any of the parameters of a staking pool.
     STAKER-17: A non-administrator (of any pool) should not be able to change any of the parameters of a staking pool,
     ↪  even if they are administrators of a different pool.
     STAKER-18: An administrator of a staking pool should be able to transfer administration of that pool to another
     ↪  account.
     STAKER-138: A non-administrator should not be able to transfer administration of a pool to another account.
     STAKER-139: A non-administrator of a staking pool should not be able to transfer administration of that pool to
     ↪  another account, even if they are an administrator of another pool.
     STAKER-19: A holder should be able to stake any number of native tokens into a native token staking position.
     STAKER-20: A holder should be able to stake any number of ERC20 tokens into an ERC20 staking position.
     STAKER-21: A holder should be able to stake an ERC721 token into an ERC721 staking position.
     STAKER-22: A holder should be able to stake any number of ERC1155 tokens into an ERC1155 staking position.
     STAKER-23: Staking position tokens for transferable staking pools should be transferable.
     STAKER-24: Staking position tokens for non-transferable staking pools should not be transferable.
     STAKER-25: If a native token staking pool does not have a cooldown, the user who staked into that position should
     ↪  be able to unstake after the lockup period assuming they still hold the position token.
     STAKER-26: If an ERC20 staking pool does not have a cooldown, the user who staked into that position should be able
     ↪  to unstake after the lockup period assuming they still hold the position token.
     STAKER-27: If an ERC721 staking pool does not have a cooldown, the user who staked into that position should be
     ↪  able to unstake after the lockup period assuming they still hold the position token.
     STAKER-28: If an ERC1155 staking pool does not have a cooldown, the user who staked into that position should be
     ↪  able to unstake after the lockup period assuming they still hold the position token.
     STAKER-29: If a native token staking pool does not have a cooldown, a user who holds the position token but who
     ↪  isn't the original holder should be able to unstake after the lockup period.
     STAKER-30: If an ERC20 staking pool does not have a cooldown, a user who holds the position token but who isn't the
     ↪  original holder should be able to unstake after the lockup period.
     STAKER-31: If an ERC721 staking pool does not have a cooldown, a user who holds the position token but who isn't
     ↪  the original holder should be able to unstake after the lockup period.
     STAKER-32: If an ERC1155 staking pool does not have a cooldown, a user who holds the position token but who isn't
     ↪  the original holder should be able to unstake after the lockup period.
```

```
    STAKER-33: If a native token staking pool does not have a cooldown, a user who doesn't hold the position token
    ↪  should not be able to unstake a position in that staking pool even after the lockup period.
    STAKER-34: If an ERC20 staking pool does not have a cooldown, a user who doesn't hold the position token should not
    ↪  be able to unstake a position in that staking pool even after the lockup period.
    STAKER-35: If an ERC721 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool even after the lockup period.
    STAKER-36: If an ERC1155 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool even after the lockup period.
    STAKER-37: If a native token staking pool does not have a cooldown, a user who doesn't hold the position token
    ↪  should not be able to unstake a position in that staking pool even after the lockup period, even if they were
    ↪  the original holder.
    STAKER-38: If an ERC20 staking pool does not have a cooldown, a user who doesn't hold the position token should not
    ↪  be able to unstake a position in that staking pool even after the lockup period, even if they were the original
    ↪  holder.
    STAKER-39: If an ERC721 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool even after the lockup period, even if they were the
    ↪  original holder.
    STAKER-40: If an ERC1155 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool even after the lockup period, even if they were the
    ↪  original holder.
    STAKER-49: If a native token staking pool does not have a cooldown, a user who doesn't hold the position token
    ↪  should not be able to unstake a position in that staking pool before the lockup period expires.
    STAKER-50: If an ERC20 staking pool does not have a cooldown, a user who doesn't hold the position token should not
    ↪  be able to unstake a position in that staking pool before the lockup period expires.
    STAKER-51: If an ERC721 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool before the lockup period expires.
    STAKER-52: If an ERC1155 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool before the lockup period expires.
    STAKER-53: If a native token staking pool does not have a cooldown, a user who doesn't hold the position token
    ↪  should not be able to unstake a position in that staking pool before the lockup period expires, even if they
    ↪  were the original holder
    STAKER-54: If an ERC20 staking pool does not have a cooldown, a user who doesn't hold the position token should not
    ↪  be able to unstake a position in that staking pool before the lockup period expires, even if they were the
    ↪  original holder
    STAKER-55: If an ERC721 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool before the lockup period expires, even if they were the
    ↪  original holder
    STAKER-56: If an ERC1155 staking pool does not have a cooldown, a user who doesn't hold the position token should
    ↪  not be able to unstake a position in that staking pool before the lockup period expires, even if they were the
    ↪  original holder.
    STAKER-57: If a native token staking pool does not have a cooldown, a user who holds the position token should not
    ↪  be able to unstake a position in that staking pool before the lockup period expires
    STAKER-58: If an ERC20 staking pool does not have a cooldown, a user who holds the position token should not be
    ↪  able to unstake a position in that staking pool before the lockup period expires
    STAKER-59: If an ERC721 staking pool does not have a cooldown, a user who holds the position token should not be
    ↪  able to unstake a position in that staking pool before the lockup period expires
    STAKER-60: If an ERC1155 staking pool does not have a cooldown, a user who holds the position token should not be
    ↪  able to unstake a position in that staking pool before the lockup period expires
    STAKER-41: If a native token staking pool has a cooldown, a position holder who did create the position and who has
    ↪  not initiated an unstake should not be able to unstake their position even after the lockup period
    STAKER-42: If an ERC20 staking pool has a cooldown, a position holder who did create the position and who has not
    ↪  initiated an unstake should not be able to unstake their position even after the lockup period
    STAKER-43: If an ERC721 staking pool has a cooldown, a position holder who did create the position and who has not
    ↪  initiated an unstake should not be able to unstake their position even after the lockup period
    STAKER-44: If an ERC1155 staking pool has a cooldown, a position holder who did create the position and who has not
    ↪  initiated an unstake should not be able to unstake their position even after the lockup period
    STAKER-45: If a native token staking pool has a cooldown, a position holder who didn't create the position and who
    ↪  has not initiated an unstake should not be able to unstake their position even after the lockup period
    STAKER-46: If an ERC20 staking pool has a cooldown, a position holder who didn't create the position and who has
    ↪  not initiated an unstake should not be able to unstake their position even after the lockup period
    STAKER-47: If an ERC721 staking pool has a cooldown, a position holder who didn't create the position and who has
    ↪  not initiated an unstake should not be able to unstake their position even after the lockup period
    STAKER-48: If an ERC1155 staking pool has a cooldown, a position holder who didn't create the position and who has
    ↪  not initiated an unstake should not be able to unstake their position even after the lockup period

  Staker
    STAKER-61: If a native token staking pool has a cooldown, a position holder who didn't create the position should
    ↪  not be able to initiate an unstake before the lockup period has expired
    STAKER-62: If an ERC20 staking pool has a cooldown, a position holder who didn't create the position should not be
    ↪  able to initiate an unstake before the lockup period has expired
    STAKER-63: If an ERC721 staking pool has a cooldown, a position holder who didn't create the position should not be
    ↪  able to initiate an unstake before the lockup period has expired
    STAKER-64: If an ERC1155 staking pool has a cooldown, a position holder who didn't create the position should not
    ↪  be able to initiate an unstake before the lockup period has expired
```

```
 STAKER-65: If a native token staking pool has a cooldown, a position holder who did create the position should not
↪  be able to initiate an unstake before the lockup period has expired
 STAKER-66: If an ERC20 staking pool has a cooldown, a position holder who did create the position should not be
↪  able to initiate an unstake before the lockup period has expired
 STAKER-67: If an ERC721 staking pool has a cooldown, a position holder who did create the position should not be
↪  able to initiate an unstake before the lockup period has expired
 STAKER-68: If an ERC1155 staking pool has a cooldown, a position holder who did create the position should not be
↪  able to initiate an unstake before the lockup period has expired
 STAKER-69: If a native token staking pool has a cooldown, a position non-holder should not be able to initiate an
↪  unstake before the lockup period has expired
 STAKER-70: If an ERC20 staking pool has a cooldown, a position non-holder should not be able to initiate an unstake
↪  before the lockup period has expired
 STAKER-71: If an ERC721 staking pool has a cooldown, a position non-holder should not be able to initiate an
↪  unstake before the lockup period has expired
 STAKER-72: If an ERC1155 staking pool has a cooldown, a position non-holder should not be able to initiate an
↪  unstake before the lockup period has expired
 STAKER-73: If a native token staking pool has a cooldown, a position non-holder should not be able to initiate an
↪  unstake after the lockup period has expired
 STAKER-74: If an ERC20 staking pool has a cooldown, a position non-holder should not be able to initiate an unstake
↪  after the lockup period has expired
 STAKER-75: If an ERC721 staking pool has a cooldown, a position non-holder should not be able to initiate an
↪  unstake after the lockup period has expired
 STAKER-76: If an ERC1155 staking pool has a cooldown, a position non-holder should not be able to initiate an
↪  unstake after the lockup period has expired
 STAKER-77: If a native token staking pool has a cooldown, a position holder should be able to initiate an unstake
↪  after the lockup period has expired
 STAKER-78: If an ERC20 staking pool has a cooldown, a position holder should be able to initiate an unstake after
↪  the lockup period has expired
 STAKER-79: If an ERC721 staking pool has a cooldown, a position holder should be able to initiate an unstake after
↪  the lockup period has expired
 STAKER-80: If an ERC1155 staking pool has a cooldown, a position holder should be able to initiate an unstake after
↪  the lockup period has expired
 STAKER-81: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
↪  unstake but not completed that unstake, any further initiations of the unstake will be idempotent
 STAKER-82: If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake but
↪  not completed that unstake, any further initiations of the unstake will be idempotent
 STAKER-83: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake but
↪  not completed that unstake, any further initiations of the unstake will be idempotent
 STAKER-84: If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an unstake
↪  but not completed that unstake, any further initiations of the unstake will be idempotent
 STAKER-85: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
↪  unstake, and if the cooldown period has not expired, then the position holder cannot complete the unstake
 STAKER-86: If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake, and
↪  if the cooldown period has not expired, then the position holder cannot complete the unstake
 STAKER-87: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
↪  and if the cooldown period has not expired, then the position holder cannot complete the unstake
 STAKER-88: If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
↪  and if the cooldown period has not expired, then the position holder cannot complete the unstake
 STAKER-89: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
↪  unstake, and if the cooldown period has not expired, then a position non-holder cannot complete the unstake
 STAKER-90: If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake, and
↪  if the cooldown period has not expired, then a position non-holder cannot complete the unstake
 STAKER-91: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
↪  and if the cooldown period has not expired, then a position non-holder cannot complete the unstake
 STAKER-92: If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
↪  and if the cooldown period has not expired, then a position non-holder cannot complete the unstake
 STAKER-93: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
↪  unstake, and if the cooldown period has not expired, then a position non-holder cannot complete the unstake,
↪  even if they were the original creator of the position
 STAKER-94: If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake, and
↪  if the cooldown period has not expired, then a position non-holder cannot complete the unstake, even if they
↪  were the original creator of the position
 STAKER-95: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
↪  and if the cooldown period has not expired, then a position non-holder cannot complete the unstake, even if they
↪  were the original creator of the position
 STAKER-96: If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
↪  and if the cooldown period has not expired, then a position non-holder cannot complete the unstake, even if they
↪  were the original creator of the position
 STAKER-97: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
↪  unstake, and if the cooldown period has expired, then the position holder can unstake and the position token is
↪  burned when the staking position is transferable
 STAKER-98: : If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
↪  and if the cooldown period has expired, then the position holder can unstake and the position token is burned
↪  when the staking position is transferable
```

```
    STAKER-99: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then the position holder can unstake and the position token is burned
    ↪  when the staking position is transferable
    STAKER-100: : If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an
    ↪  unstake, and if the cooldown period has expired, then the position holder can unstake and the position token is
    ↪  burned when the staking position is transferable
    STAKER-101: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
    ↪  unstake, and if the cooldown period has expired, then the position holder can unstake and the position token is
    ↪  burned when the staking position is non-transferable
    STAKER-102: : If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then the position holder can unstake and the position token is burned
    ↪  when the staking position is non-transferable
    STAKER-103: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then the position holder can unstake and the position token is burned
    ↪  when the staking position is non-transferable
    STAKER-104: : If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an
    ↪  unstake, and if the cooldown period has expired, then the position holder can unstake and the position token is
    ↪  burned when the staking position is non-transferable
    STAKER-105: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
    ↪  unstake, and if the cooldown period has expired, then a position non-holder cannot complete the unstake
    STAKER-106: If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then a position non-holder cannot complete the unstake
    STAKER-107: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then a position non-holder cannot complete the unstake
    STAKER-108: If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then a position non-holder cannot complete the unstake
    STAKER-109: If a native token staking pool has a cooldown, if a position holder has successfully initiated an
    ↪  unstake, and if the cooldown period has expired, then a position non-holder cannot complete the unstake, even if
    ↪  they were the original creator of the position
    STAKER-110: If an ERC20 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then a position non-holder cannot complete the unstake, even if they
    ↪  were the original creator of the position
    STAKER-111: If an ERC721 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then a position non-holder cannot complete the unstake, even if they
    ↪  were the original creator of the position
    STAKER-112: If an ERC1155 staking pool has a cooldown, if a position holder has successfully initiated an unstake,
    ↪  and if the cooldown period has expired, then a position non-holder cannot complete the unstake, even if they
    ↪  were the original creator of the position

Staker
    STAKER-113: The ERC721 representing an ERC721 staking position have as its metadata URI a data URI representing an
    ↪  appropriate JSON object
    STAKER-114: The ERC721 representing a non-ERC721 staking position have as its metadata URI a data URI representing
    ↪  an appropriate JSON object
    STAKER-115: `CurrentAmountInPool` and `CurrentPositionsInPool` should accurate reflect the amount of tokens and
    ↪  number of positions currently open under a native token staking pool
    STAKER-116 `CurrentAmountInPool` and `CurrentPositionsInPool` should accurate reflect the amount of tokens and
    ↪  number of positions currently open under an ERC20 staking pool
    STAKER-117 `CurrentAmountInPool` and `CurrentPositionsInPool` should accurate reflect the amount of tokens and
    ↪  number of positions currently open under an ERC721 staking pool
    STAKER-118: `CurrentAmountInPool` and `CurrentPositionsInPool` should accurately reflect the amount of tokens and
    ↪  number of positions currently open under an ERC1155 staking pool
    STAKER-119: `CurrentAmountInPool` and `CurrentPositionsInPool` should not be affected by positions opened under
    ↪  other pools
    `STAKER-120`: For pools without cooldowns, changes to the `lockupSeconds` setting apply to all unstaked users
    `STAKER-121`: For pools with cooldowns, for users who have not yet initiated a cooldown, changes to the
    ↪  `lockupSeconds` setting apply to determine when it is possible for them to `initiateUnstake`
    `STAKER-122`: For pools with cooldowns, for users who have initiated a cooldown already, changes to the
    ↪  `cooldownSeconds` setting apply to their final unstake
    `STAKER-123`: If an administrator changes `transferable` from `true` to `false`, position tokens are no longer
    ↪  transferable even if they were transferable, and were transferred! before
    `STAKER-124`: If an administrator changes `transferable` from `true` to `false`, position tokens that were not
    ↪  transferable before become transferable if so configured
    `STAKER-125`: Position tokens from transferable pools can be staked back into the `Staker`
    `STAKER-126`: A user must call the correct `stake*` method to stake their tokens
    `STAKER-127`: When a user calls `stakeNative`, they must stake a non-zero number of tokens
    `STAKER-128`: When a user calls `stakeERC20`, they must stake a non-zero number of tokens
    `STAKER-129`: When a user calls `stakeERC1155`, they must stake a non-zero number of tokens
    `STAKER-130`: Calls to `tokenURI` for position tokens of unstaked positions should revert
    `STAKER-131`: If a user who holds a position in a pool with `cooldownSeconds = 0` calls `initiateUnstake` after the
    ↪  lockup period has expired, the `unstakeInitiatedAt` parameter of the position is updated
    `STAKER-132`: If a user who holds a position in a pool with `cooldownSeconds = 0` calls `initiateUnstake` before
    ↪  the lockup period has expired, the transaction reverts
    `STAKER-133`: If an admin pool who want to stake native tokens for another user should successfully stake
    `STAKER-134`: If an admin pool who want to stake native tokens for another user should successfully stake - non
    ↪  transferable pool
    `STAKER-135`: If an admin pool who want to stake erc20 tokens for another user should successfully stake
    `STAKER-136`: If an admin pool who want to stake erc20 tokens for another user should successfully stake - non
    ↪  transferable pool
    `STAKER-137`: If a staker user who want to stake erc721 tokens for another user should successfully stake
    `STAKER-138`: If a staker user who want to stake erc721 tokens for another user should successfully stake - non
    ↪  transferable pool
    `STAKER-139`: If a staker user who want to stake erc1155 tokens for another user should successfully stake
    `STAKER-140`: If a staker user who want to stake erc1155 tokens for another user should successfully stake - non
    ↪  transferable
    `STAKER-141`: As an admin pool I can unstake any position in the pool that still active
    `STAKER-142`: As an admin pool I can unstake any position in the pool that still active even if the pool - non
    ↪  transferable
    `STAKER-143`: As an admin pool unstake should works even if the position holder were transferred
    `STAKER-144`: As an admin pool I can unstake any position the owner of the position should recieve back the native
    ↪  tokens
    `STAKER-145`: As an admin pool I can unstake any position the owner of the position should recieve back the erc20
    ↪  tokens
    `STAKER-146`: As an admin pool I can unstake any position the owner of the position should recieve back the erc721
    ↪  tokens
    `STAKER-147`: As an admin pool I can unstake any position the owner of the position should recieve back the native
    ↪  erc1155 tokens
    STAKER-148: Any account should be able to create a staking pool for native tokens using 0 address
```

8.3 Automated Tools

8.3.1 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

8.3.2 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. 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 procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, 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