# Security Review Report
# NM-0534 Lagoon

(May 13, 2025)

# Contents

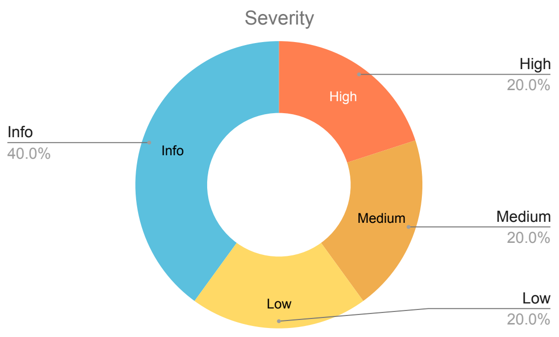
# 1 Executive Summary

This document presents the security review performed by Nethermind Security for Lagoon factory contract. Lagoon Protocol is a decentralized asset management platform that enables asset managers to create Lagoon Vaults. These Vaults provide efficient, non-custodial, and risk-managed asset management solutions.

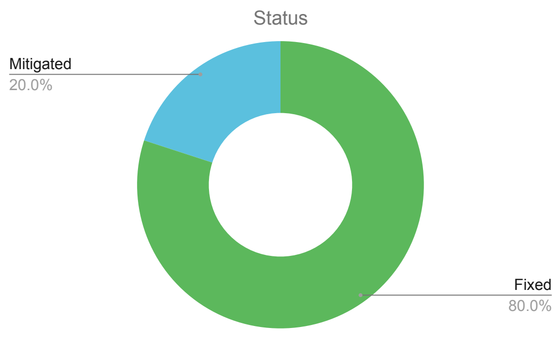
Built on a foundation of smart contract standards, Lagoon Protocol leverages the power of Gnosis Safe, Zodiac Roles Modifier, and other key components to create highly customizable and secure vaults for managing digital assets.

**The audit focuses** on the PR #200 which introduces new features like synchronous deposits and a burn functionality. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report five points of attention**, where one is classified as `High`, one is classified as `Medium`, one is classified as `Low`, and 2 are classified as `Informational` as shown 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 compilation, tests, and automated tests. Section 9 concludes the document.



(a)



(b)

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

### Summary of the Audit

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | May 9, 2025 |
| **Final Report** | May 13, 2025 |
| **Repository** | lagoon-v0 |
| **Commit** | 84e43bf30c1b0698b978ed37ec184c5f4957b715 |
| **Final Commit** | ca04ed2b51baf1a75ed6c7d64f1e5e6c656aed16 |
| **Documentation Assessment** | Medium |
| **Test Suite Assessment** | Medium |

## 2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | src/v0.5.0/Roles.sol | 70 | 36 | 51.4% | 13 | 119 |
| 2 | src/v0.5.0/Whitelistable.sol | 61 | 23 | 37.7% | 11 | 95 |
| 3 | src/v0.5.0/Silo.sol | 14 | 4 | 28.6% | 5 | 23 |
| 4 | src/v0.5.0/FeeManager.sol | 146 | 73 | 50.0% | 43 | 262 |
| 5 | src/v0.5.0/ERC7540.sol | 447 | 170 | 38.0% | 129 | 746 |
| 6 | src/v0.5.0/Vault.sol | 313 | 137 | 43.8% | 62 | 512 |
| 7 | src/v0.5.0/primitives/Errors.sol | 22 | 33 | 150.0% | 26 | 81 |
| 8 | src/v0.5.0/primitives/Events.sol | 34 | 50 | 147.1% | 22 | 106 |
| 9 | src/v0.5.0/primitives/Enums.sol | 6 | 2 | 33.3% | 3 | 11 |
| 10 | src/v0.5.0/primitives/Struct.sol | 16 | 12 | 75.0% | 4 | 32 |
| 11 | src/v0.5.0/interfaces/IERC7540.sol | 9 | 2 | 22.2% | 5 | 16 |
| 12 | src/v0.5.0/interfaces/IWETH9.sol | 8 | 4 | 50.0% | 3 | 15 |
| 13 | src/v0.5.0/interfaces/IERC7540Deposit.sol | 16 | 8 | 50.0% | 7 | 31 |
| 14 | src/v0.5.0/interfaces/IERC7540Redeem.sol | 10 | 1 | 10.0% | 5 | 16 |
| 15 | src/v0.5.0/interfaces/IERC7575.sol | 5 | 2 | 40.0% | 2 | 9 |
| | **Total** | **1177** | **557** | **47.3%** | **340** | **2074** |

***Only features introduced in PR 200**.

## 3 Summary of Issues

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Native token deposits are lost if synchronous deposits are available | High | Fixed |
| 2 | `NAV` proposals may disregard increases to `totalAssets` from syncrhonous deposits | Medium | Mitigated |
| 3 | Share price can be artificially inflated through synchronous deposits coupled with share burning | Low | Fixed |
| 4 | The `requestDeposit(...)` function is inconsistent depending on the executed branch | Info | Fixed |
| 5 | `syncDeposit(...)` function lacks operator and referral features present in asynchronous deposits | Info | Fixed |

# 4 System Overview

The recent protocol modifications introduce notable changes to existing vault processes and enable new avenues for user interaction. Key updates include:

- **syncDeposit(...):** This new function facilitates synchronous deposits. The availability of this feature relies on the time of the last settlement and `totalAssetsLifespan` state variable; synchronous deposits are enabled immediately following a NAV settlement and remain permissible for the duration specified by `totalAssetsLifespan`. This mechanism allows users to deposit assets at the prevailing share price and receive their corresponding shares without delay. Upon a synchronous deposit, the contributed assets are transferred into the safe, and the vault's `totalAssets` variable is increased to reflect this addition.
- **requestDeposit(...)**: The behavior of this existing function has been altered. If synchronous deposits are currently permissible (i.e., within the active window defined by `totalAssetsLifespan` post-settlement), `requestDeposit(...)` will now preferentially execute an immediate synchronous deposit rather than creating an asynchronous deposit request.
- **burn(...)**: This is a newly introduced function that grants users the ability to destroy their vault shares. Burning shares reduces the `totalSupply` of shares in circulation.

# 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 [High] Native token deposits are lost if synchronous deposits are available

**File(s)**: `src/v0.5.0/Vault.sol`

**Description**: The `requestDeposit(...)` function is designed to handle deposits, including those made with native tokens (e.g., ETH). A recent modification causes `requestDeposit(...)` to fallback to an internal synchronous deposit mechanism, `_syncDeposit(...)`, if synchronous deposits are enabled in the vault.

The vulnerability arises because `_syncDeposit(...)` is exclusively designed to handle `ERC20` token deposits and does not account for `msg.value` if the initial call to `requestDeposit(...)` was made with native tokens.

The relevant internal `_syncDeposit(...)` logic is:

```
function _syncDeposit(...) internal onlyOpen returns (uint256 shares) {
    ERC7540Storage storage $ = _getERC7540Storage();

    shares = _convertToShares(assets, Math.Rounding.Floor);
    $.totalAssets += assets;
    SafeERC20.safeTransferFrom(IERC20(asset()), owner, safe(), assets);
    _mint(receiver, shares);

    emit DepositSync(msg.sender, receiver, assets, shares);
}
```

If a user tries to execute a native token deposit through `requestDeposit(...)` and synchronous deposits are available the native tokens sent wont be taken into account through the deposit operation.

**Recommendation(s)**: Consider modifying `_syncDeposit(...)` to manage native token deposits.

**Status**: Fixed.

**Update from the client**: We added the native token deposit to the `syncDeposit(...)` function.

**Update from Nethermind Security**: The issue is fixed in commit 85ab53.

## 6.2 [Medium] `NAV` proposals may disregard increases to `totalAssets` from syncrhonous deposits

**File(s)**: `src/v0.5.0/ERC7540.sol`

**Description**: The `totalAssets` variable within the `ERC7540` contract, which is critical for share price calculation, is updated through two distinct mechanisms:

- NAV Settlements: During a settlement, the `totalAssets` variable is set to the `newTotalAssets` value proposed by the NAV committee based on their off-chain valuation;
- Synchronous Deposits: The introduction of synchronous deposits allows `totalAssets` to be incremented directly upon a successful deposit, reflecting the immediate inflow of new assets;

A race condition vulnerability exists due to the potential timing mismatch between these two update paths. If the NAV committee performs a valuation and submits a `newTotalAssets` proposal before recent synchronous deposits are made (or before they are aware of them), this proposed value will not reflect the assets added by those deposits.

When a settlement subsequently occurs, the `totalAssets` variable will be overwritten with the NAV committee's proposed (and now outdated) value. This action effectively negates the `totalAssets` increments from any synchronous deposits that occurred between the NAV committee's valuation time and the settlement execution. While the shares minted from these synchronous deposits remain valid, the `totalAssets` backing them is temporarily unrepresented in the contract's state, leading to a deflated share price.

**Recommendation(s)**: To address this vulnerability, it is crucial to ensure that any proposed new valuation accurately incorporates the impact of all synchronous deposits made up to the valuation point. Two primary approaches can be considered:

- **Operational Safeguards:** Managers could implement a procedural control by delaying the computation of a new NAV until the period for synchronous deposits relevant to that valuation cycle has definitively closed. This would ensure the valuation is based on the most current asset totals. However, it's important to note that such operational measures cannot be enforced at the smart contract level with the existing implementation and rely on consistent off-chain adherence.
- **Contract-Level Modification (Delta-Based Updates):** A more robust solution involves modifying how `totalAssets` is updated during NAV settlements. Instead of overwriting `totalAssets` with an absolute value proposed by the NAV committee, the system could be designed to apply a delta (i.e., the change in value of the managed assets, excluding synchronous deposits already accounted for on-chain). This approach would ensure that increases to `totalAssets` from synchronous deposits are preserved, as the NAV committee's reported change would be added to the already updated `totalAssets` figure, thus preventing the inadvertent exclusion of these deposited assets.

**Status**: Mitigated.

**Update from the client**: In order to solve this issue, we opted for a solution close to the operationnal safeguards. We offer the vault.safe() role (curator) the capacity to expire `totalAssets`. By doing so he will disable the capacity to do synchronous deposit until next complete totalAssets update, offering the ValuationManager and himself the liberty to update the valuation without risks.

**Update from Nethermind Security**: The issue is mitigated at commit ca04ed.

## 6.3 [Low] Share price can be artificially inflated through synchronous deposits coupled with share burning

**File(s)**: src/v0.5.0/Vault.sol

**Description**: The vault's share price is implicitly determined by the ratio `totalAssets / totalSupply`. TThe interaction between the two recently introduced features allow users to inflate the share's price:

- Synchronous Deposits: Whitelisted users can directly deposit assets into the vault. This action increases the `totalAssets` value and mints a corresponding number of shares to the depositor;
- Share Burning: The `burn(...)` function allows users to destroy a specified amount of their own shares, thereby decreasing `totalSupply`;

A malicious (whitelisted) user can exploit this by performing the following sequence:

- Deposit Assets: The attacker deposits a quantity of assets ( `X`) into the vault. This increases `totalAssets` by `X` and mints `Y` shares for the attacker;
- Burn Shares: The attacker then immediately calls the `burn(...)` function to destroy all or a significant portion of the `Y` shares they just received;

The consequence is that the `X` assets contributed by the attacker remain within the vault (increasing `totalAssets`, but the corresponding shares that claim those assets are removed from circulation (decreasing `totalSupply`). This effectively acts as a "donation" of the attacker's deposited assets to the remaining pool, concentrating the vault's total asset value over fewer shares and thereby inflating the price of each remaining share.

Artificially inflating the share price can lead to several adverse consequences. Malicious actors could exploit this vulnerability, particularly early in the vault's operation or when total share supply is low, to conduct "first deposit attacks"; this would cause subsequent legitimate depositors to receive disproportionately fewer shares for their assets, unfairly benefiting the attacker. Furthermore, if the vault's share price is utilized as an oracle by other DeFi protocols for functions like collateral valuation, this manipulation could be leveraged to deceive or exploit those external systems for an attacker's gain. Such potential for manipulation can also undermine user confidence and deter legitimate participation in the vault.

**Recommendation(s)**: Reconsider the addition of the shares burning feature. The removal of this feature would remove the current possibility of donations through synchronous deposits and burn.

**Status**: Fixed.

**Update from the client**: We removed the capacity to burn shares.

**Update from Nethermind Security**: Fixed in commit d05006.

## 6.4 [Info] The `requestDeposit(...)` function is inconsistent depending on the executed branch

**File(s)**: `src/v0.5.0/Vault.sol`

**Description**: The `requestDeposit(...)` function has been modified to serve a dual purpose: it initiates an asynchronous deposit request or, if synchronous deposits are currently available, it falls back to executing a synchronous deposit directly. This conditional branching within a single function leads to several inconsistencies in its behavior and outputs, making it more difficult for developers and integrating systems to reliably predict and handle its execution.

The observed inconsistencies include:

- Return Value Discrepancy:
  - Asynchronous Path: When initiating an asynchronous deposit, `requestDeposit(...)` returns the unique request ID associated with the pending operation.
  - Synchronous Path: If the function executes a synchronous deposit, it returns 0 (zero), as no asynchronous request ID is generated. Callers must be prepared to handle these two distinct return value meanings.
- Referral Processing:
  - Asynchronous Path: Referral parameters are processed and accounted for.
  - Synchronous Path: Referral parameters are ignored, and no referral logic is triggered.
- Handling of Existing Pending Requests:
  - Asynchronous Path: The function is expected to fail if the caller already has a pending asynchronous deposit request.
  - Synchronous Path: If a synchronous deposit is possible, the function may proceed with the synchronous deposit, potentially bypassing or altering the logic that checks for and handles existing pending asynchronous requests for the user. This means the pre-condition regarding pending requests is not consistently enforced.

**Recommendation(s)**: Consider maintaining `requestDeposit(...)` exclusively for asynchronous deposit requests and `syncDeposit(...)` for synchronous deposits.

**Status**: Fixed.

**Update from the client**: We gave up the "overload" of the requests functions. They will remain as before, and synchronous deposits are only possible by calling the function `syncDeposit(...)`.

**Update from Nethermind Security**: Fixed in commit d05006.

## 6.5 [Info] `syncDeposit(...)` function lacks operator and referral features present in asynchronous deposits

**File(s)**: `src/v0.5.0/Vault.sol`

**Description**: The recently introduced `syncDeposit(...)` function enables users to perform synchronous deposits when this operational mode is active. This provides an immediate deposit pathway, contrasting with the existing asynchronous `requestDeposit(...)` function.

However, `syncDeposit()` currently does not implement two features available in the asynchronous deposit flow:

- Operator Feature: The asynchronous `requestDeposit(...)` allows for an owner parameter, distinct from msg.sender. This enables an approved operator ( `msg.sender`) to initiate a deposit request on behalf of the owner of the assets. The `syncDeposit(...)` function lacks an owner parameter; assets are taken directly from `msg.sender`, meaning `msg.sender` is always the owner of the deposited assets. Thus, the pattern of an operator depositing assets on behalf of another distinct owner is not supported;
- Referral Feature: The `requestDeposit(...)` function includes a referral parameter, allowing for on-chain tracking and potential incentivization of referrals. This parameter is absent in the `syncDeposit(...)` function signature and its associated logic;

This difference means that users and integrated systems cannot utilize operator-based deposits or referral tracking when using the synchronous deposit method.

- Native Token Deposits: The asynchronous deposit mechanism may support deposits of the chain's native token (e.g., ETH). The `syncDeposit(...)` function, as currently implemented, appears to be primarily designed for `ERC20` token deposits and lacks a clear mechanism or explicit handling for native token deposits;

**Recommendation(s)**: To promote consistency and broader utility, consider aligning the feature set of `syncDeposit(...)` more closely with `requestDeposit(...)`.

**Status**: Fixed.

**Update from the client**: We added the referral system to the `syncDeposit(...)` function and use 0 as the *requestId*. We decided not to support the "operator" feature in synchronous deposits.

**Update from Nethermind Security**: The Lagoon team has added the desired features. The issue is fixed at commit ca04ed.

# 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 the Lagoon contract documentation**

The Lagoon team explained the use cases and expected behavior of the newly added features.

# 8 Complementary Checks

## 8.1 Compilation Output

```
forge build
[] Compiling...
[] Compiling 284 files with Solc 0.8.26
[] Solc 0.8.26 finished in 103.95s
Compiler run successful with warnings:
Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:201:9:
    |
201 |         _deposit(_msgSender(), receiver, assets, shares);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:203:9:
    |
203 |         return shares;
    |         ^^^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:218:9:
    |
218 |         _deposit(_msgSender(), receiver, assets, shares);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:220:9:
    |
220 |         return assets;
    |         ^^^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:231:9:
    |
231 |         _withdraw(_msgSender(), receiver, owner, assets, shares);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:233:9:
    |
233 |         return shares;
    |         ^^^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:244:9:
    |
244 |         _withdraw(_msgSender(), receiver, owner, assets, shares);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (5740): Unreachable code.
   --> lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol:246:9:
    |
246 |         return assets;
    |         ^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> test/v0.5.0/SyncDepositViaRequestDeposit.t.sol:30:9:
   |
30 |         uint256 ret = vault.requestDeposit(userBalanceBefore, user1.addr, user1.addr);
   |         ^^^^^^^^^^^
```

## 8.2 Tests Output

```
forge test --match-path "**/Sync**"
[] Compiling...
No files changed, compilation skipped

Ran 7 tests for test/v0.5.0/SyncDeposit.t.sol:TestSyncDeposit
[PASS] test_syncDeposit() (gas: 178748)
[PASS] test_syncDeposit_addressZeroReceiver() (gas: 283135)
[PASS] test_syncDeposit_differentReceiver() (gas: 340682)
[PASS] test_syncDeposit_lifespanOutdate() (gas: 36465)
[FAIL. Reason: Error != expected error: 0x4b0145f30000000000000000000000000000000000000000000000000000000000000001 !=
↪  0x4b0145f30000000000000000000000000000000000000000000000000000000000000002] test_syncDeposit_whenClosed() (gas:
↪  285085)
[PASS] test_syncDeposit_whenPaused() (gas: 313768)
[PASS] test_syncDeposit_whitelist() (gas: 161758)
Suite result: FAILED. 6 passed; 1 failed; 0 skipped; finished in 5.43s (204.81ms CPU time)

Ran 7 tests for test/v0.5.0/SyncDepositViaRequestDeposit.t.sol:TestSyncDeposit
[PASS] test_syncDeposit() (gas: 181834)
[PASS] test_syncDeposit_addressZeroReceiver() (gas: 285125)
[PASS] test_syncDeposit_differentReceiver() (gas: 342688)
[PASS] test_syncDeposit_lifespanOutdate() (gas: 150432)
[PASS] test_syncDeposit_whenClosed() (gas: 275068)
[PASS] test_syncDeposit_whenPaused() (gas: 301377)
[PASS] test_syncDeposit_whitelist() (gas: 162800)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 5.43s (285.16ms CPU time)

Ran 2 test suites in 35.82s (10.85s CPU time): 13 tests passed, 1 failed, 0 skipped (14 total tests)

Failing tests:
Encountered 1 failing test in test/v0.5.0/SyncDeposit.t.sol:TestSyncDeposit
[FAIL. Reason: Error != expected error: 0x4b0145f30000000000000000000000000000000000000000000000000000000000000001 !=
↪  0x4b0145f30000000000000000000000000000000000000000000000000000000000000002] test_syncDeposit_whenClosed() (gas:
↪  285085)

Encountered a total of 1 failing tests, 13 tests succeeded
```

## 8.3 Automated Tools

### 8.3.1 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 process, from initial concepts to successful implementation.

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

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

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

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

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

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

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

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

**Learn more about us at nethermind.io.**

### General Advisory to Clients

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

### Disclaimer

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