Security Review Report
NM-0304 Lagoon Protocol

(Nov 7, 2024)

Contents

1 Executive Summary

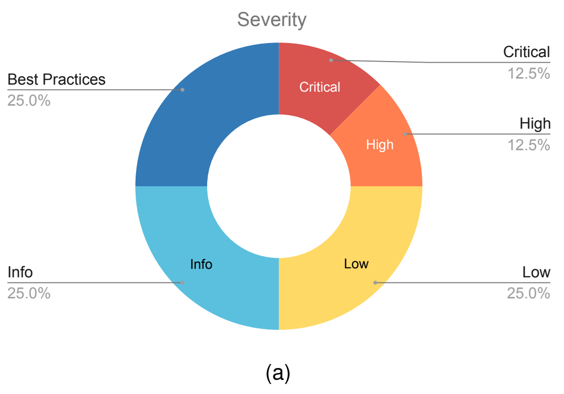
This document presents the security review performed by Nethermind Security for Lagoon Protocol smart contracts. Lagoon Protocol is a decentralized asset management platform that enables asset managers to create Lagoon Vaults. These Vaults provide efficient, non-custodial, and risk-managed asset management 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 audited code comprises 1056 lines of code written in the Solidity language. The audit focuses on the implementation of **ERC-7540 Vault** contract.

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** eight points of attention, where one is classified as `Critical`, one is classified as `High`, two are classified as `Low`, and four are as `Informational` or `Best Practice`. 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 compilation, tests, and automated tests. Section 9 concludes the document.



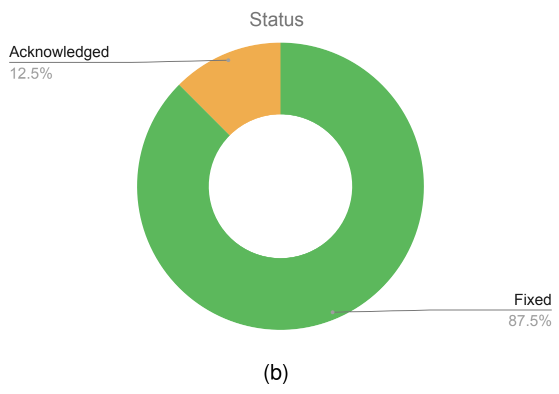


Fig. 1: Distribution of issues: Critical (1), **High** (1), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (2), **Best Practices** (2).
Distribution of status: Fixed (7), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	October 18, 2024
Final Report	November 7, 2024
Repository	lagoon-v0
Commit	090461aff6aca5f42f3013a51679c62ed9fe78c3
Final Commit	cd6135f5b948b5b6c32abfc2201bd2d1b96221f4
Documentation	Docs
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/protocol/Events.sol	4	11	275.0%	3	18
2	src/protocol/FeeRegistry.sol	57	35	61.4%	14	106
3	src/vault/Roles.sol	62	36	58.1%	13	111
4	src/vault/Whitelistable.sol	51	23	45.1%	11	85
5	src/vault/Silo.sol	14	4	28.6%	5	23
6	src/vault/FeeManager.sol	145	67	46.2%	42	254
7	src/vault/ERC7540.sol	415	173	41.7%	129	717
8	src/vault/Vault.sol	191	91	47.6%	39	321
9	src/vault/primitives/Errors.sol	20	31	155.0%	24	75
10	src/vault/primitives/Events.sol	31	39	125.8%	19	89
11	src/vault/primitives/Enums.sol	6	2	33.3%	3	11
12	src/vault/primitives/Struct.sol	14	12	85.7%	4	30
13	src/vault/interfaces/IERC7540.sol	9	2	22.2%	5	16
14	src/vault/interfaces/IWETH9.sol	6	4	66.7%	3	13
15	src/vault/interfaces/IERC7540Deposit.sol	16	8	50.0%	7	31
16	src/vault/interfaces/IERC7540Redeem.sol	10	1	10.0%	5	16
17	src/vault/interfaces/IERC7575.sol	5	2	40.0%	2	9
	Total	**1056**	**541**	**51.2%**	**328**	**1925**

3 Summary of Issues

	Finding	Severity	Update
1	The user can mint vault shares without providing the correct amount of underlying native assets	Critical	Fixed
2	Pending assets are moved to the vault while still in a pending state	High	Fixed
3	The `_withdraw(...)` and `_mint(...)` functions do not round in favor of the protocol	Low	Fixed
4	The `highWatermark` is set inconsistently	Low	Unresolved
5	Shares can be burned when depositing to the `address(0)`	Info	Fixed
6	`DOS` attacks on controllers are possible between settlements	Info	Acknowledged
7	Implementation contracts can be initialized	Best Practices	Fixed
8	The `msg.sender` is accessed inconsistently	Best Practices	Fixed

4 System Overview

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.

Lagoon Protocol enables the creation of decentralized Vaults (Lagoon Vaults) that support various roles, including Asset Managers, NAV Committees, Vault Creators, and Fund Depositors. These Vaults are governed by smart contracts that allow for a wide range of DeFi strategies, from asset management to yield farming, all while maintaining a high level of security and control. The protocol's design prioritizes flexibility, enabling asset managers to configure their Vaults with specific DeFi protocol whitelists, separation of power, fee structures, and more. It follows the **ERC-7540** standard for **Asynchronous ERC-4626 Tokenized Vaults**.

4.1 Actors

- **AssetManager:** The asset manager is the actor in charge of using the funds in the Vault to generate more value. The AssetManager will interact with the Vault through a Gnosis Safe wallet that will restrict the type of operations the manager can do. Only the manager can execute settlement operations.
- **NAV Committees:** The NAV Committe will submit new valuation proposals to the Vault. This actor is in charge of calculating the current value of all the assets held by the Vault and submitting it for future settlement.
- **Owner:** The owner of the Vault can change multiple parameters from the configuration of the vault, including the fee rates and receivers. The owner can also change the current NAV Committee address or start the Vault closing procedure.
- **Users:** The User actor refers to regular users using the Vault. These users will deposit assets in exchange for shares that can be redeemed later. Redeeming shares will give the user a certain amount of assets relative to the valuation of the vault at the moment of the redeeming action.

4.2 Vault States

A Vault can be in three main states: **Open**, **Closing**, and **Closed**. Depending on the current state of the Vault, it will behave differently for certain actions.

- **Open:** After a Vault is initialized, it will be automatically in the **Open** state. In this state, the Vault is open to deposits and withdrawals. New valuations can be proposed and settled. An **Open** Vault can only transition to the **Closing** state.
- **Closing:** Only the **Onwer** actor can change the Vault to the **Closing** state. The *requestRedeem* operations are no longer possible in this state. It is possible for the NAV Committee to propose new valuations, but settlement operations cannot be executed. To execute another settlement, a call to the *close(...)* function must be made; this call will transition the Vault to the **Closed** state.
- **Closed:** At this state, the Vault is closed; settlements are locked, and withdrawals are guaranteed at a fixed price per share. No new valuations can be proposed, and settlements cannot be executed anymore. The Vault cannot transition to a different state.

It is important to note that **requestDeposit** operations are always available, even when executing any new settlements is impossible. If a user requests a deposit and no new settlement can be executed, the user can cancel the request and recover the funds.

4.3 Deposit and Redeem operations

Deposit and Redemption are the most important operations from the Vault. These are the actions executed by users to deposit assets and redeem shares. These operations are executed asynchronously. Users first need to create a request and wait until a settlement is executed, which will complete their request. After a request is completed, a user can finish the deposit or redeem operation. A request can only be completed if it was placed before the final evaluation for the current settlement period. This means that if the settlement occurs after the request has been placed but without any evaluations between these two events, this settlement will not complete the request. The user will have to wait until the next evaluation and settlement.

Compared to synchronous vaults, in asynchronous vaults, the users do not know the price per share that will be executed for their operation at the moment of the request. When they start a request, the involved assets are sent to the pending silo contract until the operation is settled. After the operation is settled, depending on the type of operation, the funds will be sent to the asset manager or the vault. Users can claim their assets or shares based on the price per share related to their settlement.

4.4 Fees

The Vault will charge two types of fees: management and performance fees.

The management fees are calculated based on the assets under management (AUM) and are charged over time and collected during vault settlement. The formula used here calculates management fees by multiplying the assets by the management rate (a percentage expressed in basis points or BPS) and prorating it by the time elapsed (relative to one year).

$$managementFee = (\frac{asset * rate}{BPS}) * \frac{time_elapsed}{1year}$$

The performance fees are charged on profits and are calculated only when the value of the assets exceeds a high water mark (the highest historical value per one share). This is done to ensure that fees are charged only on actual profits and not on recovered losses.

$$performanceFee = \frac{((pricePerShare - highWaterMark) * totalSupply) * rate}{BPS}$$

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 user can mint vault shares without providing the correct amount of underlying native assets

File(s): src/vault/ERC7540.sol

Description: **This issue was found and reported by the Lagoon Protocol team**. Whenever the users request a deposit into the Vault contract, they specify the amount of assets they wish to deposit. If the Vault's asset is configured to be a wrapped native token, the users can send the native tokens as msg.value in the call to requestDeposit(...) function and the vault will wrap the tokens for them automatically. The problem present in the _requestDeposit(...) function is that it assumes that the assets specified by the user match the provided msg.value, which might not be the case.

```
function _requestDeposit(uint256 assets, address controller, address owner)
    internal
    returns (uint256)
{
    // ...
    // @audit `assets` amount is user controlled. Internal accounting
    // accepts whatever was provided by the user.
    $.epochs[_depositId].depositRequest[controller] += assets;

    if (msg.value != 0) {
        if (asset() == address($.wrappedNativeToken)) {
            // @audit-issue The user can specify arbitrary `assets` amount
            // and provide fewer native tokens. The request will go through.
            // After that, the user can finish the deposit and mint vault shares,
            // even though not enough underlying native tokens were provided.
            $.pendingSilo.depositEth{value: msg.value}();
        } else {
            revert CantDepositNativeToken();
        }
    } else {
        IERC20(asset()).safeTransferFrom(
            owner, address($.pendingSilo), assets
        );
    }
    // ...
}
```

In this scenario, the user can specify an arbitrary amount of assets without sending that amount of native tokens to the contract. The provided assets amount is saved in the internal accounting and will be used to compute the shares the user will receive upon deposit completion.

The shares transferred to the user on deposit completion are minted beforehand once the deposits are settled in _settleDeposit(...). The number of shares minted depends on the actual token balances of the PendingSilo contract. Since the user could have provided fewer native tokens, fewer shares would have been minted as a result.

However, on deposit completion, the user will be rewarded with shares computed based on internal accounting, which is higher than expected. There might be a situation where the Vault contract will attempt to transfer more shares to the user than it currently holds and revert. The other scenario is more severe. The Vault might have enough shares to cover the user's deposit at the cost of other users.

Recommendation(s): Ensure the asset parameter is equal to msg.value when using native token.

Status: Fixed.

Update from the client: The issue was fixed in this commit.

6.2 [High] Pending assets are moved to the vault while still in a pending state

File(s): `src/vault/Vault.sol`

Description: Whenever a user requests an action, the assets/shares are moved to the `pendingSilo` contract. Then, after a settlement operation, they are moved out from the `pendingSilo` contract. For example, if a user requests a deposit, the assets will be sent to the `pendingSilo`. After that, when `settleDeposit(...)` is called, the funds will be sent from the `pendingSilo` to the asset manager, and the new shares for the requested assets will be minted.

A problem arises because all the assets in the `pendingSilo` are moved and used to settle the operation, even though there may be some assets that should remain pending. These assets are the ones related to users who requested their action before the settlement but after the last valuation update that occurred. This behavior has several unwanted impacts.

As a first impact, the users who request a deposit between the last valuation update and the settlement will not be able to cancel their deposit, even though they should be able to. This is due to the funds not being in the `pendingSilo` anymore.

Secondly, if the previous impact is analyzed in the context of closing the vault, the users would lose their funds, as they will never be able to complete their request or cancel it.

As a third impact, this situation will create an inconsistency in the amount of existing shares. This is because the new shares will be minted in the settlement operation at a specific price per share. However, the user can only execute the deposit at the price per share of the next settlement. If the price per share increases between the two settlements, there will be a remainder of the shares that cannot be claimed. On the other hand, if the price per share decreases between the two settlements, there will not be enough shares to cover the requested deposit from all the users.

Recommendation(s): Consider only moving from the `pendingSilo` and executing the request that will move to a completed state.

Status: Fixed.

Update from the client: Fixed in this commit.

6.3 [Low] The _withdraw(...) and _mint(...) functions do not round in favor of the protocol

File(s): src/vault/ERC7540.sol

Description: As explained in the EIP4626, it is considered most secure to always round in favor of the vault. This is not the case in the _withdraw(...) and _mint(...) functions where the rounding favors the user.

The _withdraw(...) function receives the amount of assets the user wants to withdraw and computes the amount of shares that must be given in return. The computation of how many shares must be given rounds down.

```
function _withdraw(uint256 assets, address receiver, address controller) internal returns (uint256 shares) {
    ERC7540Storage storage $ = _getERC7540Storage();

    uint40 requestId = $.lastRedeemRequestId[controller];
    if (requestId > $.lastRedeemEpochIdSettled) {
        revert RequestIdNotClaimable();
    }

    // @audit - The computation of the shares rounds down
    shares = convertToShares(assets, requestId);
    $.epochs[requestId].redeemRequest[controller] -= shares;
    IERC20(asset()).safeTransfer(receiver, assets);

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

The _mint(...) function receives the amount of shares to be minted and computes how many assets the user must transfer. The computation how many assets the user must transfer rounds down.

```
function _mint(uint256 shares, address receiver, address controller) internal virtual returns (uint256 assets) {
    ERC7540Storage storage $ = _getERC7540Storage();

    uint40 requestId = $.lastDepositRequestId[controller];
    if (requestId > $.lastDepositEpochIdSettled) {
        revert RequestIdNotClaimable();
    }

    // @audit - The computation of how many assets must be transferred rounds down
    assets = _convertToAssets(shares, requestId, Math.Rounding.Floor);

    $.epochs[requestId].depositRequest[controller] -= assets;
    _update(address(this), receiver, shares);

    emit Deposit(controller, receiver, assets, shares);
}
```

Recommendation(s): Consider changing rounding directions to always favor the Vault.

Status: Fixed.

Update from the client: Fixed in this commit.

6.4 [Low] The `highWatermark` is set inconsistently

File(s): `src/vault/Vault.sol`

Description: The `highWatermark` value is used to compute the `performanceFee`. This value represents the highest price of one share. Every time a settlement occurs, the price per share is computed, and if it is greater than the previously stored `highWatermark` the new value is stored.

```
function _setHighWaterMark(uint256 _newHighWaterMark) internal {
    FeeManagerStorage storage $ = _getFeeManagerStorage();

    uint256 _highWaterMark = $.highWaterMark;

    if (_newHighWaterMark > _highWaterMark) {
        emit HighWaterMarkUpdated(_highWaterMark, _newHighWaterMark);
        $.highWaterMark = _newHighWaterMark;
    }
}
```

The fees are computed in the `_calculateFees(...)` function. If the new price per share is higher than the current `highWatermark` value, this function will compute the current price per share and calculate the performance fee. Since fees are paid in shares, new shares will be minted to accommodate them. As a result of the newly minted shares, the share price in the vault will automatically decrease after the fees are charged.

The `_setHighWaterMark(...)` function is called right after that. However, the price per share used to compute the fees in the previous step will differ from that used for the `_setHighWaterMark(...)`, creating an inconsistency. The price used in the `_setHighWaterMark(...)` function is the already decreased share price.

Recommendation(s): Consider calling the `_setHighWaterMark(...)` function with the price of the shares before the fees were charged.

Status: Unresolved

Update from the client: We made a first implementation inside this commit and then refactored inside this commit. The difference is that the `pricePerShare` is return from `_calculateFees()` instead of being passed to it as a parameter.

6.5 [Info] Shares can be burned when depositing to the `address(0)`

File(s): `src/vault/ERC7540.sol`

Description: To receive the shares for an asset deposit in the asynchronous vault, the deposit controller has to call the `deposit(...)` function on the `Vault` contract. The caller controls the `receiver` address that will receive the shares. Internally, the `deposit(...)` function uses the `_update(...)` function from OpenZeppelin's `ERC20` to update the receiver address's shares. An issue arises when the controller accidentally specifies the receiver as an `address(0)`. Since the `_update(...)` function treats transfers to the `address(0)` as burns, the vault share tokens would be burned.

Recommendation(s): It is considered a best practice to use higher-level `ERC20` functions that use `_update(...)` internally and better capture the caller's intent, such as `_transfer(...)`, `_mint(...)`, and `_burn(...)`. Consider using `_transfer(...)` instead of `_update(...)` to transfer the shares to the `receiver` address during the deposit.

Status: Fixed.

Update from the client: Fixed in this commit.

6.6 [Info] `DOS` attacks on controllers are possible between settlements

File(s): `src/vault/ERC7540.sol`

Description: `controllers` can only have one request for `redeem` or `deposit` in the `Pending` state. If a `redeem` or `deposit` request is created for a controller at epoch `x`, no other `redeem` or `deposit` request can be created for the same controller while the previous one is in the `Pending` state.

Because any user can create a request associated with any controller, it is possible for a user to create a request for a controller with a minimum amount. As a result, the controller won't be able to create new requests until a settlement occurs.

It is important to note that until the epoch changes, requests can be modified by the controller without creating a new one.

Recommendation(s): Consider documenting the previously described behavior. To reduce possible damages caused by this, consider frequently executing settlement operations.

Status: Acknowledged.

Update from the client: We won't fix this.

6.7 [Best Practices] Implementation contracts can be initialized

File(s): `src/vault/Vault.sol`, `src/protocol/FeeRegistry.sol`

Description: The implementation contracts `Vault` and `FeeRegistry` do not disable the `initialize(...)` functions. This allows anyone to call `initialize(...)` on these contracts with arbitrary values, which may lead to unwanted behavior. An example of such malicious behavior is initiating a call from the implementation contract to a sanctioned or black-listed contract such as Tornado Cash. This action could lead to unnecessary legal issues for institutions that use the protocol smart contracts.

Recommendation(s): Consider removing the possibility of calling the `initialize(...)` functions on the implementation contracts. This may be done by calling the `_disableInitializers(...)` in the constructor, which will disable calling functions with the `initializer` modifier.

Status: Fixed.

Update from the client: Fixed in commit and commit.

6.8 [Best Practices] The `msg.sender` is accessed inconsistently

File(s): `src/vault/Vault.sol`

Description: The `Vault` contract's `redeem(...)` function calls the `_msgSender(...)` function from OpenZeppelin's `Context` module to get the address of an account that initiated the redemption. This is unnecessary since the caller in the current execution context can be accessed directly from the `msg.sender` property. Usage of `_msgSender(...)` may lead to inconsistencies in the future.

Recommendation(s): Consider removing an unnecessary call to `_msgSender(...)` and replace it with direct access to the `msg.sender` to make the code more readable.

Status: Fixed.

Update from the client: Fixed in commit.

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 Protocol documentation

The documentation for the Lagoon Protocol was provided through their official docs. This documentation provided a high-level overview of the protocol and details of its implementation. Moreover, the Lagoon Protocol team addressed all 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

Lagoon protocol's test suite is comprehensive and covers all the core flows of the protocol. To further enhance it and determine potential uncovered code paths, the Nethermind Security team applied a technique called mutation testing. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch and "kill" the mutant.

The following table outlines the results of the analysis performed on the Lagoon protocol's core contracts. The first column lists the contracts that were tested using mutation testing. The second column indicates the number of mutants generated and how many were "slain" (i.e., caught by the test suite). The third column provides the percentage of mutants slain, reflecting the effectiveness of the test suite in covering the particular contract.

Contract	Mutants (slain / total generated)	Score
`ERC7540`	261/285	91.57%
`FeeManager`	131/142	92.25%
`FeeRegistry`	8/8	100%
`Silo`	3/3	100%
`Vault`	57/68	83.82%
`Whitelistable`	17/19	89.47%

The following is a list of code modifications not caught by the existing test suite. Each point highlights a scenario that could benefit from higher test coverage to enhance the protocol's overall security and resilience in the next code change iterations as the project evolves.

ERC7540.sol

- The expression `_totalSupply += shares` inside the `_settleDeposit(...)` function is only used to emit an event. The value by which the `_totalSupply` is incremented can be arbitrarily changed or even set to `0` with no effect on the test suite. The same happens inside the `_settleRedeem(...)` function. An additional test can be added to check that the value emitted in the event is exactly what it is expected to be.
- The expression `$erc7540.redeemSettleId = redeemSettleId + 2` inside the `_settleRedeem(...)` function can be entirely removed from the codebase with no effect on the test suite. An additional test can be added to check that the `redeemSettleId` before and after redemption settlements are different and are changed as expected.
- The function `pendingDepositRequest(...)` is not tested for cases where `requestID` is smaller than `lastDepositEpochIdSettled`. In these situations, the default value of `0` is returned since there are no pending deposit requests after settlement. A test case can be added to ensure that the state of the contract reflects this expectation.
- The assignment `requestId = $.lastDepositRequestId[controller]` in the `claimableDepositRequest(...)` function can be replaced with a hardcoded `requestID` of `1` with no effect on the test suite. The tests can be extended with end-to-end scenarios where a single controller goes through the whole protocol flow multiple times to test that the request IDs are updated correctly and point to appropriate asset values.
- No test case ensures that the user's deposit request was set to `0` after the `cancelRequestDeposit(...)` was called.

FeeManager.sol

- In the assignment `managementFee = annualFee.mulDiv(...)` in the `_calculateManagementFee(...)` function, `managementFee` can be assigned a hardcoded `0` value instead, with no effect on the test suite. Test cases can be added to ensure the computed `managementFee` matches the expectations.
- No test case targets the early return in the `_takeFees(...)` function. A test case can be added to ensure that the shares are not minted twice if redemptions are settled before the deposits. Additional test cases can be added to test scenarios where the `managerShares` are minted but not the `protocolShares`.

Vault.sol

- No test case would trigger the `NotWhitelisted` error. A test case can be added to ensure that a non-whitelisted user cannot call the `requestDeposit(...)` function (the one with four parameters including the referral; the other `requestDeposit(...)` is tested).

- In the `settleRedeem(...)` function, the call to the `_takeFees(...)` function can be entirely removed with no effect on the test suite. Test cases can be added around the amounts of specific fees minted during the redemption settlement. Such tests are also missing for the `close(...)` function.
- The `settleRedeem(...)` function's tests are not checking that the new high watermark was actually set. The `_setHighWaterMark(...)` can be called with an arbitrary value, and the tests will pass (e.g. setting the watermark to `1`, while `1000` was used to determine whether to take the performance fee or not).
- In the `close(...)` function, calls to the `_settleDeposit(...)` and `_settleRedeem(...)` functions can be removed with no effect on the test suite. Tests can be added to ensure that the vault is correctly settled before closing it.

8.1 Compilation Output

```
> forge build --force
[] Compiling...
[] Compiling 119 files with Solc 0.8.26
[] Solc 0.8.26 finished in 21.24s
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 (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> src/vault/ERC7540.sol:181:98:
    |
181 |     function previewDeposit(uint256) public pure override(ERC4626Upgradeable, IERC4626) returns (uint256 shares) {
    |                                                                                                  ^^^^^^^^^^^^^^
```

8.2 Tests Output

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

Ran 10 tests for test/Roles.t.sol:TestMint
[PASS] test_feeReceiver() (gas: 12896)
[PASS] test_feeRegistry() (gas: 12832)
[PASS] test_protocolFeeReceiver() (gas: 18180)
[PASS] test_safe() (gas: 12809)
[PASS] test_updateFeeReceiver() (gas: 19988)
[PASS] test_updateNewTotalAssetsManager() (gas: 19942)
[PASS] test_updateNewTotalAssetsManager_notOwner() (gas: 11274)
[PASS] test_updateWhitelistManager() (gas: 19813)
[PASS] test_valuationManager() (gas: 12830)
[PASS] test_whitelistManager() (gas: 12760)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 2.92s (1.88ms CPU time)

Ran 2 tests for test/Silo.t.sol:TestSilo
[PASS] test_constructorGivesInfiniteApprovalToMsgSender() (gas: 135957)
[PASS] test_vaultHasInfiniteApprovalOnPendingSilo() (gas: 25739)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 3.24s (323.46ms CPU time)

Ran 1 test for test/Referral.t.sol:TestReferral
[PASS] test_referral() (gas: 155062)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.66s (278.00ms CPU time)

Ran 3 tests for test/CancelRequest.t.sol:TestCancelRequest
[PASS] test_cancelRequestDeposit() (gas: 122079)
[PASS] test_cancelRequestDeposit_revertsWhenNewTotalAssetsHasBeenUpdated() (gas: 178810)
[PASS] test_cancelRequestDeposit_when0PendingRequest() (gas: 17823)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 4.12s (5.49ms CPU time)

Ran 3 tests for test/Deposit.t.sol:TestDeposit
[PASS] test_deposit() (gas: 601380)
[PASS] test_deposit_revertIfNotOperator() (gas: 15744)
[PASS] test_deposit_revertIfRequestIdNotClaimable() (gas: 152454)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 4.12s (738.73ms CPU time)

Ran 4 tests for test/Operator.t.sol:TestOperator
[PASS] test_addOperator() (gas: 44395)
[PASS] test_addOperatorwhenOpIsAlreadyOp() (gas: 48579)
[PASS] test_rmvOperator() (gas: 34688)
[PASS] test_rmvOperatorWhenAddressIsNotOperator() (gas: 24515)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 460.24ms (4.63ms CPU time)

Ran 4 tests for test/Mint.t.sol:TestMint
[PASS] test_mint() (gas: 648118)
[PASS] test_mintAsOperator() (gas: 672212)
[PASS] test_mint_revertIfNotOperator() (gas: 15741)
[PASS] test_mint_revertIfRequestIdNotClaimable() (gas: 152450)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 4.12s (745.72ms CPU time)

Ran 6 tests for test/unitTests/FeeRegistry.t.sol:TestFeeRegistry
[PASS] test_cancelCustomRate() (gas: 44037)
[PASS] test_customRate() (gas: 57969)
[PASS] test_init() (gas: 12721)
[PASS] test_protocolRate() (gas: 30010)
[PASS] test_updateProtocolFeeReceiver() (gas: 20787)
[PASS] test_updateProtocolFeeReceiver_revertIfNotOwner() (gas: 10736)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 2.32ms (686.88µs CPU time)

Ran 15 tests for test/Misc.t.sol:TestMisc
[PASS] test_contractSize() (gas: 4944727)
[PASS] test_decimals() (gas: 26949)
[PASS] test_depositId() (gas: 610477)
[PASS] test_epochSettleId() (gas: 960926)
[PASS] test_getRoleStorage() (gas: 32675)
[PASS] test_lastDepositRequestId() (gas: 987751)
[PASS] test_lastRedeemRequestId() (gas: 1536878)
[PASS] test_pendingSilo() (gas: 26168)
[PASS] test_previewDeposit() (gas: 8518)
```

```
[PASS] test_previewMint() (gas: 8521)
[PASS] test_previewRedeem() (gas: 8467)
[PASS] test_previewWithdraw() (gas: 8544)
[PASS] test_redeemId() (gas: 857867)
[PASS] test_share() (gas: 8778)
[PASS] test_supportsInterface() (gas: 19479)
Suite result: ok. 15 passed; 0 failed; 0 skipped; finished in 4.13s (753.61ms CPU time)

Ran 4 tests for test/Redeem.t.sol:TestRedeem
[PASS] test_redeem() (gas: 889688)
[PASS] test_redeem_revertIfRequestIdNotClaimable() (gas: 665830)
[PASS] test_redeem_whenNotOperatorShouldRevert() (gas: 861828)
[PASS] test_redeem_whenOperator() (gas: 921306)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 4.13s (747.01ms CPU time)

Ran 13 tests for test/Pause.t.sol:TestPause
[PASS] test_cancelRequestDeposit_whenPaused_shouldFail() (gas: 158240)
[PASS] test_deposit_whenPaused_shouldFail() (gas: 54220)
[PASS] test_mint_whenPaused_shouldFail() (gas: 53648)
[PASS] test_pauseShouldPause() (gas: 10724)
[PASS] test_requestDeposit_whenPaused_shouldFail() (gas: 21174)
[PASS] test_requestRedeem_whenPaused_shouldFail() (gas: 23284)
[PASS] test_setOperator_whenPaused_shouldFail() (gas: 13505)
[PASS] test_settleDeposit_whenPaused_shouldFail() (gas: 86628)
[PASS] test_settleRedeem_whenPaused_shouldFail() (gas: 86583)
[PASS] test_unpauseShouldUnpause() (gas: 14726)
[PASS] test_updateNewTotalAssets_whenPaused_shouldFail() (gas: 19484)
[PASS] test_withdraw_whenPausedAndVaultClosed_shouldFail() (gas: 505566)
[PASS] test_withdraw_whenPaused_shouldFail() (gas: 331409)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 9.02ms (3.13ms CPU time)

Ran 3 tests for test/Withdraw.t.sol:TestWithdraw
[PASS] test_withdraw() (gas: 899733)
[PASS] test_withdraw_revertIfNotOperator() (gas: 20180)
[PASS] test_withdraw_revertIfRequestIdNotClaimable() (gas: 674906)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 9.85ms (5.12ms CPU time)

Ran 7 tests for test/RequestRedeem.t.sol:TestRequestRedeem
[PASS] test_requestRedeem() (gas: 110620)
[PASS] test_requestRedeemTwoTimes() (gas: 124142)
[PASS] test_requestRedeem_OnlyOneRequestAllowed() (gas: 163469)
[PASS] test_requestRedeem_asAnOperator() (gas: 150180)
[PASS] test_requestRedeem_asAnOperatorNotAllowed() (gas: 31056)
[PASS] test_requestRedeem_notEnoughBalance() (gas: 74613)
[PASS] test_requestRedeem_withClaimableBalance() (gas: 442944)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 15.62ms (2.51ms CPU time)

Ran 13 tests for test/Whitelist.t.sol:TestWhitelist
[PASS] test_addToWhitelist_revert() (gas: 7032919)
[PASS] test_noWhitelist() (gas: 6842897)
[PASS] test_requestDeposit_ShouldFailWhenControllerNotWhitelisted() (gas: 7035708)
[PASS] test_requestDeposit_ShouldNotFailWhenControllerNotWhitelistedandOperatorAndOwnerAre() (gas: 7126182)
[PASS] test_requestDeposit_WhenOwnerWhitelistedAndOperator() (gas: 7150466)
[PASS] test_requestRedeemWithoutBeingWhitelisted() (gas: 7734012)
[PASS] test_revokeFromWhitelist_revert() (gas: 7032953)
[PASS] test_transfer_ShouldWorkWhenReceiverWhitelisted() (gas: 7577215)
[PASS] test_transfer_WhenReceiverNotWhitelistedAfterDeactivateOfWhitelisting() (gas: 7534920)
[PASS] test_unwhitelist() (gas: 7067939)
[PASS] test_unwhitelistList() (gas: 7066818)
[PASS] test_whitelist() (gas: 7059449)
[PASS] test_whitelistList() (gas: 7087839)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 906.30ms (920.23ms CPU time)

Ran 14 tests for test/InitiateClosing.t.sol:TestInitiateClosing
[PASS] test_CloseCantBeCalledAfterVaultIsClosed() (gas: 116418)
[PASS] test_canNotCallInitiateClosingTwice() (gas: 16051)
[PASS] test_cantCloseAVaultWithoutFullUnwind() (gas: 128213)
[PASS] test_cantUpdateNewTotalAssetsWhenClosed() (gas: 120049)
[PASS] test_closingVaultMarkTheVaultAsClosed() (gas: 115225)
[PASS] test_inClosedStateCanWithdrawAndRedeemIfOperatorOrEnoughAllowance() (gas: 298477)
[PASS] test_inClosingStateCanNotWithdrawOrRedeemIfNotOperatorAndEvenWithEnoughAllowance() (gas: 64107)
[PASS] test_inClosingStateCanWithdrawAndRedeemIfOperator() (gas: 106728)
[PASS] test_newSettleDepositAreForbiddenButClaimsAreAvailable() (gas: 189266)
```

```
[PASS] test_redeemAssetWithoutClaimableRedeem() (gas: 209967)
[PASS] test_redeemSharesWithClaimableRedeem() (gas: 338312)
[PASS] test_redeemSharesWithClaimableRedeemWithProfits() (gas: 417776)
[PASS] test_requestRedemptionAreImpossible() (gas: 152971)
[PASS] test_withdrawAssetWithoutClaimableRedeem() (gas: 208665)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 4.25s (141.06ms CPU time)

Ran 6 tests for test/FeeManager.t.sol:TestFeeManager
[PASS] test_FeesAreTakenAfterFreeride() (gas: 2385728)
[PASS] test_NoFeesAreTakenDuringFreeRide() (gas: 1961428)
[PASS] test_defaultHighWaterMark_equalsPricePerShares() (gas: 20211)
[PASS] test_feeReceiverAndDaoHaveNoVaultSharesAtVaultCreation() (gas: 25799)
[PASS] test_updateRates_revertIfManagementRateAboveMaxRates() (gas: 24047)
[PASS] test_updateRates_revertIfPerformanceRateAboveMaxRates() (gas: 24115)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 4.38s (1.47s CPU time)

Ran 12 tests for test/RequestDeposit.t.sol:TestRequestDeposit
[PASS] test_only_one_request_allowed_per_settle_id() (gas: 220006)
[PASS] test_requestDeposit() (gas: 160631)
[PASS] test_requestDepositTwoTimes() (gas: 198529)
[PASS] test_requestDeposit_ShouldBeAbleToDepositAgainWhenIndeterminationIsRaidedAtSettlement() (gas: 645565)
[PASS] test_requestDeposit_asAnOperator() (gas: 203693)
[PASS] test_requestDeposit_asAnOperatorButOwnerNotEnoughApprove() (gas: 115260)
[PASS] test_requestDeposit_asAnOperatorNotAllowed() (gas: 33724)
[PASS] test_requestDeposit_notEnoughBalance() (gas: 98581)
[PASS] test_requestDeposit_revertIfNotOperator() (gas: 15763)
[PASS] test_requestDeposit_withClaimableBalance() (gas: 693639)
[PASS] test_requestDeposit_withClaimableBalance_with_eth() (gas: 3332)
[PASS] test_requestDeposit_with_eth() (gas: 6985905)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 1.57s (1.67s CPU time)

Ran 5 tests for test/RatesUpdate.t.sol:testRateUpdates
[PASS] test_ratesShouldMatchValuesAtInit() (gas: 6767593)
[PASS] test_ratesShouldRevertAtInitWhenToHigh() (gas: 6747650)
[PASS] test_updateRatesOverMaxPerformanceRateShouldRevert() (gas: 6765861)
[PASS] test_updateRatesShouldBeApplyed24HoursAfter() (gas: 6802029)
[PASS] test_updateRatesShouldBeApplyed24HoursAfter_VerifyThroughASettle() (gas: 7745652)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 4.96s (3.60s CPU time)

Ran 10 tests for test/Settle.t.sol:TestSettle
[PASS] test_close_revertIfNotTotalAssetsManager() (gas: 12115)
[PASS] test_settleDepositAfterUpdate() (gas: 302979)
[PASS] test_settleDepositThenRedeemAfterUpdate() (gas: 123777)
[PASS] test_settleDeposit_revertIfNotTotalAssetsManager() (gas: 12203)
[PASS] test_settleRedeemAfterUpdate() (gas: 266607)
[PASS] test_settleRedeem_revertIfNotTotalAssetsManager() (gas: 12183)
[PASS] test_settle_deposit_without_totalAssets_update_reverts() (gas: 7341842)
[PASS] test_settle_redeem_totalAssets_update_reverts() (gas: 7473651)
[PASS] test_simple_settle() (gas: 590681)
[PASS] test_updateNewTotalAssets_revertIfNotTotalAssetsManager() (gas: 12230)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 5.39s (2.37s CPU time)

Ran 19 test suites in 5.88s (52.40s CPU time): 135 tests passed, 0 failed, 0 skipped (135 total tests)
```

9 About Nethermind

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

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

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

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

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

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

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

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

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

Learn more about us at nethermind.io.

General Advisory to Clients

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

Disclaimer