Security Review Report
NM-0856 - Royco Makina Strategy



NETHERMIND
SECURITY

(March 16, 2026)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 The Royco Vault and Strategy Interface . . . 4
4.2 The Makina Machine Integration . . . 4
4.3 Access Control and Safety Mechanisms . . . 4

5 Risk Rating Methodology **5**

6 Issues **6**
6.1 [Low] Large withdrawals spanning multiple strategies revert when Makina strategy hits machine liquidity limit . . . 6
6.2 [Info] The `onWithdraw(...)` function redeems one share when called with zero amount . . . 7
6.3 [Info] The `rescueToken(...)` function does not emit an event as required by the `IStrategyTemplate` interface . . . 8
6.4 [Info] The `rescueToken(...)` function only checks against `MACHINE_SHARE_TOKEN` instead of `asset()` as specified by the `IStrategyTemplate` interface . . . 9
6.5 [Info] `maxWithdraw()` does not account for paused state or Machine recovery mode . . . 10
6.6 [Best Practices] Undocumented calldata structure for allocation and deallocation parameters . . . 10

7 Documentation Evaluation **11**

8 Test Suite Evaluation **12**
8.1 Automated Tools . . . 12
8.1.1 AuditAgent . . . 12

9 About Nethermind **13**

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for Royco Protocol's smart contracts.

The Royco Protocol team has developed a strategy adapter contract (`RoycoVaultMakinaStrategy`) that bridges their Concrete Earn v2 vault system with Makina Finance's yield-generating Machine infrastructure. The strategy acts as a thin integration layer: the Concrete vault allocates idle assets into the strategy, which deposits them into a dedicated Makina Machine, and redeems Machine shares back into assets when the vault needs to service user withdrawals or administrative deallocations.

The contract implements the `IStrategyTemplate` interface, exposing `allocateFunds(...)`, `deallocateFunds(...)`, and `onWithdraw(...)` as its core operational entry points, all restricted to calls from the Royco vault. Administrative functions such as `pause()`, `unpause()`, and `rescueToken(...)` are gated by OpenZeppelin's `AccessManaged` pattern, delegating permission checks to an external authority contract. The strategy requires a dedicated Makina Machine where it is configured as the sole depositor and redeemer, enabling it to call `Machine.deposit(...)` and `Machine.redeem(...)` directly.

The review examined the strategy contract in the context of its three-way interaction surface: the upstream Concrete vault that manages user deposits and withdrawals, the strategy itself that translates between the vault's asset-denominated interface and the Machine's share-denominated model, and the downstream Makina Machine that custodies assets and deploys them to yield-generating Calibers.

The audit comprises 118 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report six points of attention, where one is classified as `Low` and five are classified as `Informational` or `Best Practices` severity. The issues are summarized in Fig. 1.

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

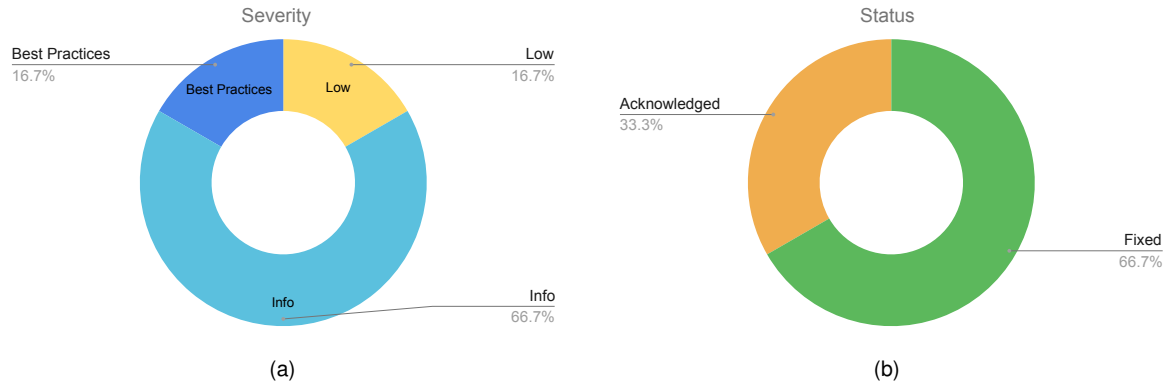


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (4), **Best Practices** (1). **Distribution of status: Fixed** (4), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	March 12, 2026
Final Report	March 16, 2026
Initial Commit	e472acc029668676940fe306de5877a60f3147c8
Final Commit	0f229b34c7f8a7a31a33bfc0bb4d43e73dabe1e8
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/RoycoVaultMakinaStrategy.sol	118	86	72.9%	40	244
	Total	**118**	**86**	**72.9%**	**40**	**244**

3 Summary of Issues

	Finding	Severity	Update
1	Large withdrawals spanning multiple strategies revert when Makina strategy hits machine liquidity limit	Low	Fixed
2	The `onWithdraw(...)` function redeems one share when called with zero amount	Info	Acknowledged
3	The `rescueToken(...)` function does not emit an event as required by the `IStrategyTemplate` interface	Info	Fixed
4	The `rescueToken(...)` function only checks against `MACHINE_SHARE_TOKEN` instead of `asset()` as specified by the `IStrategyTemplate` interface	Info	Acknowledged
5	`maxWithdraw()` does not account for paused state or Machine recovery mode	Info	Fixed
6	Undocumented calldata structure for allocation and deallocation parameters	Best Practices	Fixed

4 System Overview

The `RoycoVaultMakinaStrategy` contract is an adapter that connects Royco Protocol's ERC-4626 vaults (built on Concrete Earn V2) with Makina Protocol's yield-generating Machines. Its purpose is to enable a Royco vault to allocate its assets into a Makina Machine's cross-chain investment strategy and to redeem shares from that Machine when assets need to be returned. The strategy holds no idle funds of its own; assets are either held by the Royco vault or deposited into the Makina Machine, with the strategy acting solely as the intermediary that facilitates movement between the two.

4.1 The Royco Vault and Strategy Interface

Royco vaults are ERC-4626 compliant token vaults that can distribute their assets across one or more pluggable strategies. Each strategy must implement the `IStrategyTemplate` interface, which defines a standard set of functions for fund management and accounting. The `RoycoVaultMakinaStrategy` contract implements this interface with three core operations:

- **Allocation**: The vault calls `allocateFunds(...)` to transfer a specified amount of the base asset into the strategy. The strategy immediately deposits these assets into the Makina Machine, receiving share tokens in return. The caller provides the desired deposit amount and a minimum acceptable number of shares as slippage protection.
- **Deallocation**: The vault calls `deallocateFunds(...)` to instruct the strategy to redeem a specified number of Machine shares. The underlying assets are sent directly from the Machine to the vault. The caller specifies the number of shares to redeem and a minimum acceptable amount of assets to receive.
- **User Withdrawal**: When a vault user initiates a withdrawal, the vault calls `onWithdraw(...)` with the amount of assets required. The strategy converts this asset amount into the equivalent number of Machine shares using the Machine's current exchange rate, adds a one-share buffer to account for rounding, and redeems those shares. The resulting assets are transferred directly to the vault for distribution to the user.

The strategy also exposes several view functions consumed by the vault for rebalancing and accounting purposes. `totalAllocatedValue()` reports the current asset value of the strategy's Machine share position. `maxAllocation()` returns the maximum amount of assets the Machine can currently accept, and `maxWithdraw()` returns the lesser of the Machine's available liquidity and the strategy's own position value.

4.2 The Makina Machine Integration

The Makina Machine is the core on-chain component of the Makina Protocol. It manages deposits, redemptions, and share price calculation for a given investment strategy. Each Machine maintains its own share token, which represents proportional ownership in the Machine's total assets under management (AUM). The share price is derived from the ratio between the total share supply and the last computed total AUM, with a computed decimal offset to mitigate donation-style inflation attacks.

The strategy interacts with the Machine through two permissioned entry points: `deposit(...)` and `redeem(...)`. The Machine enforces a singleton access model for these operations — only one designated `depositor` address may call `deposit(...)`, and only one designated `redeemer` address may call `redeem(...)`. For the strategy to function, it must be registered as both the depositor and the redeemer on the target Machine via the Machine's administrative functions, `setDepositor(...)` and `setRedeemer(...)`.

During construction, the strategy grants the Machine a maximum ERC-20 approval over the base asset. This is safe under the assumption that the strategy never holds idle asset tokens — assets only pass through the strategy transiently during an allocation, moving from the vault to the Machine within a single transaction. On the redemption side, the Machine burns share tokens directly from the strategy's balance and transfers the corresponding assets to the specified receiver (the vault), requiring no separate approval step.

4.3 Access Control and Safety Mechanisms

The strategy employs several layers of access control and operational safeguards:

- **Vault-Only Operations**: The three fund management functions (`allocateFunds(...)`, `deallocateFunds(...)`, and `onWithdraw(...)`) are restricted to the Royco vault address, which is set as an immutable parameter at construction. No other address can trigger asset movements.
- **Admin-Managed Access**: Administrative functions are protected by OpenZeppelin's `AccessManaged` pattern, with the Royco factory contract serving as the access manager. This governs the ability to pause and unpause the strategy as well as to invoke the token rescue function.
- **Pausability**: Authorized admins can pause the strategy, which disables all allocation, deallocation, and withdrawal operations. The `rescueToken(...)` function remains callable while the strategy is paused, ensuring that accidentally sent tokens can still be recovered during an emergency.
- **Token Rescue**: The `rescueToken(...)` function allows admins to recover ERC-20 tokens that may have been inadvertently sent to the strategy. To protect the strategy's accounting integrity, it explicitly prevents the rescue of the Machine's share token, as those shares represent the strategy's position in the Machine and underpin the `totalAllocatedValue()` calculation.
- **Asset Compatibility Check**: The constructor enforces that the Royco vault's base asset and the Makina Machine's accounting token are identical, preventing misconfiguration at deployment time.

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 [Low] Large withdrawals spanning multiple strategies revert when Makina strategy hits machine liquidity limit

File(s): src/RoycoVaultMakinaStrategy.sol

Description: When a user withdraws from the Concrete vault and the vault's idle balance is insufficient, the vault iterates through strategies in `deallocationOrder` and calls `onWithdraw(...)` to pull assets:

```
function _executeWithdraw(...) internal virtual returns (uint256) {
    // ...
    for (uint256 i; i < strategiesCounter; ++i) {
        // ...
        uint256 withdrawableAmountFromStrategy = IStrategyTemplate(deallocationOrder[i]).maxWithdraw();
        uint256 withdrawAmount =
            (withdrawableAmountFromStrategy >= desiredAssets) ? desiredAssets : withdrawableAmountFromStrategy;

        if (withdrawAmount > 0) {
            uint256 actualWithdrawn = IStrategyTemplate(deallocationOrder[i]).onWithdraw(withdrawAmount);
            // ...
        }
    }
    // ...
}
```

The vault calculates `withdrawAmount = min(desiredAssets, strategy.maxWithdraw())`, meaning it requests exactly `maxWithdraw()` whenever the user needs more than what the strategy reports as available.

The strategy's `maxWithdraw()` correctly accounts for machine liquidity:

```
function maxWithdraw() external view returns (uint256) {
    uint256 maxWithdrawableAssets = IMachine(MAKINA_MACHINE).maxWithdraw();
    return Math.min(maxWithdrawableAssets, _getStrategyOwnedAssets());
}
```

However, the strategy's `onWithdraw(...)` function adds `+1` to the calculated shares to ensure rounding doesn't cause the user to receive less than requested:

```
function onWithdraw(uint256 _amountToWithdraw) external ... returns (uint256 amountWithdrawn) {
    // @audit-issue The +1 padding can cause sharesToRedeem to exceed machine's liquidity.
    uint256 sharesToRedeem = IMachine(MAKINA_MACHINE).convertToShares(_amountToWithdraw) + 1;
    sharesToRedeem = Math.min(sharesToRedeem, _getStrategyOwnedShares());

    amountWithdrawn = IMachine(MAKINA_MACHINE).redeem(sharesToRedeem, ROYCO_VAULT, _amountToWithdraw);
    // ...
}
```

The `Math.min(...)` inside `onWithdraw()` only caps shares at the strategy's owned balance - it does not re-validate against the machine's available liquidity after the `+1` is added. The machine's `maxWithdraw()` function returns the accounting token balance held by the machine contract. When funds are deployed to Calibers (yield-generating strategies), this balance is reduced while the strategy's share value (based on total AUM) remains unchanged. This creates a scenario where `machine.maxWithdraw()` is less than the strategy's total position value.

In this case, `strategy.maxWithdraw()` returns `machine.maxWithdraw()` (the limiting factor), and the vault calls `onWithdraw(...)` with exactly that amount. The strategy then calculates `sharesToRedeem = convertToShares(machine.maxWithdraw()) + 1`, causing `convertToAssets(sharesToRedeem)` to exceed `machine.maxWithdraw()` by approximately one share's worth. The machine's `redeem(...)` then reverts with `ExceededMaxWithdraw`.

Since the vault's `_executeWithdraw(...)` has no try-catch around the `onWithdraw(...)` call, the entire withdrawal transaction reverts. The vault does not fall back to the next strategy in `deallocationOrder`. This means that when a user requests a large withdrawal that requires pulling funds from multiple strategies, if the Makina strategy is in the deallocation sequence and hits this boundary condition, the entire withdrawal fails - even if subsequent strategies had sufficient liquidity to cover the remaining amount.

This is a common scenario when most vault funds are allocated to strategies and the underlying Makina machine has deployed funds to yield-generating Calibers, reducing its liquid balance while maintaining the same total AUM.

Users can work around this by requesting slightly less than `maxWithdraw()`, and administrators can resolve it by calling `deallocateFunds(...)` directly (which does not have the `+1` padding). However, these workarounds are not obvious and users will experience unexpected failed transactions.

Recommendation(s): Consider capping `sharesToRedeem` based on machine liquidity in addition to owned shares. Alternatively, document this behavior clearly so integrators and users are aware of the limitation.

Status: Fixed

Update from the client: Fixed

6.2 [Info] The `onWithdraw(...)` function redeems one share when called with zero amount

File(s): `src/RoycoVaultMakinaStrategy.sol`

Description: The `onWithdraw(...)` function is called by the Concrete vault to withdraw assets from the strategy when the vault's idle balance is insufficient to fulfill a user's withdrawal request. The function calculates the number of shares to redeem by converting the requested asset amount to shares and adding `1` to handle rounding:

```
function onWithdraw(uint256 _amountToWithdraw) external ... returns (uint256 amountWithdrawn) {
    // @audit-issue When `_amountToWithdraw` is zero, this evaluates to 0 + 1 = 1.
    uint256 sharesToRedeem = IMachine(MAKINA_MACHINE).convertToShares(_amountToWithdraw) + 1;
    sharesToRedeem = Math.min(sharesToRedeem, _getStrategyOwnedShares());

    amountWithdrawn = IMachine(MAKINA_MACHINE).redeem(sharesToRedeem, ROYCO_VAULT, _amountToWithdraw);
    // ...
}
```

When `_amountToWithdraw` is `0`, the `convertToShares(...)` function returns `0`, resulting in `sharesToRedeem = 1`. This causes one share to be redeemed from the Makina machine, returning its equivalent asset value to the vault despite no assets being requested. The test suite explicitly covers this scenario in `test_onWithdraw_withZeroAmount()`, acknowledging that "some shares may be redeemed" with a zero input.

Currently, this edge case is not triggered in practice because the Concrete vault only calls `onWithdraw(...)` when `withdrawAmount > 0`. However, if this strategy is reused in a different context or integrated with other systems that do not perform this check, it could lead to unintended share redemptions.

Recommendation(s): Consider adding an early return when `_amountToWithdraw` is zero, or documenting this behavior clearly in the function's NatSpec to prevent misuse in future integrations.

Status: Acknowledged

Update from the client: Acknowledged

6.3 [Info] The `rescueToken(...)` function does not emit an event as required by the `IStrategyTemplate` interface

File(s): src/RoycoVaultMakinaStrategy.sol

Description: The `IStrategyTemplate` interface defines the `rescueToken(...)` function as a mechanism for authorized admins to recover tokens that may have been accidentally sent to the strategy. The interface specification includes a MUST-level requirement stating that the function "MUST emit appropriate events for the rescue operation."

```
function rescueToken(address token, uint256 amount) external;
// - MUST emit appropriate events for the rescue operation.
```

However, the `rescueToken(...)` function in the `RoycoVaultMakinaStrategy` contract does not emit any event upon successfully rescuing tokens:

```
function rescueToken(
    address _token,
    uint256 _amount
) external override(IStrategyTemplate) restricted {
    require(
        _token != MACHINE_SHARE_TOKEN,
        INVALID_TOKEN_TO_RESCUE()
    );

    _amount = _amount == 0
        ? IERC20(_token).balanceOf(address(this))
        : _amount;
    // @audit-issue No event is emitted after the rescue.
    IERC20(_token).safeTransfer(msg.sender, _amount);
}
```

Recommendation(s): Consider defining and emitting a dedicated event (e.g., `TokenRescued(address token, uint256 amount, address recipient)`) within the `rescueToken(...)` function to comply with the `IStrategyTemplate` interface specification and improve off-chain observability of admin operations.

Status: Fixed

Update from the client: Fixed

6.4 [Info] The `rescueToken(...)` function only checks against `MACHINE_SHARE_TOKEN` instead of `asset()` as specified by the `IStrategyTemplate` interface

File(s): src/RoycoVaultMakinaStrategy.sol

Description: The `IStrategyTemplate` interface defines the `rescueToken(...)` function as a mechanism for authorized admins to recover tokens that may have been accidentally sent to the strategy. The interface specification includes MUST-level requirements for which token should be blocked from rescue:

```
function rescueToken(address token, uint256 amount) external;
// - MUST only allow rescue of tokens that are not the strategy's
//   primary asset (asset()).
// - MUST revert if attempting to rescue the strategy's primary
//   asset token.
```

However, the `rescueToken(...)` function in the `RoycoVaultMakinaStrategy` contract checks against `MACHINE_SHARE_TOKEN` (the Makina Machine's share token) rather than `ASSET` (the base/accounting token returned by `asset()`):

```
function rescueToken(
    address _token,
    uint256 _amount
) external override(IStrategyTemplate) restricted {
    // @audit Checks only against MACHINE_SHARE_TOKEN, not ASSET.
    require(
        _token != MACHINE_SHARE_TOKEN,
        INVALID_TOKEN_TO_RESCUE()
    );

    _amount = _amount == 0
        ? IERC20(_token).balanceOf(address(this))
        : _amount;
    IERC20(_token).safeTransfer(msg.sender, _amount);
}
```

Recommendation(s): Consider aligning the check with the `IStrategyTemplate` specification by also blocking `ASSET` from rescue

Status: Acknowledged

Update from the client: Acknowledged

6.5 [Info] `maxWithdraw()` does not account for paused state or Machine recovery mode

File(s): src/RoycoVaultMakinaStrategy.sol

Description: The `maxWithdraw()` function reports available liquidity without checking whether the strategy is paused or whether the underlying Makina Machine is in recovery mode. Both of these conditions cause `onWithdraw(...)` to revert, but `maxWithdraw()` does not reflect them:

```
function maxWithdraw() external view override(IStrategyTemplate) returns (uint256) {
    // @audit No check for paused() or Machine recovery mode.
    uint256 maxWithdrawableAssets = IMachine(MAKINA_MACHINE).maxWithdraw();
    return Math.min(maxWithdrawableAssets, _getStrategyOwnedAssets());
}
```

The `onWithdraw(...)` function has a `whenNotPaused` modifier, and internally calls `Machine.redeem(...)` which has a `notRecoveryMode` modifier:

```
// Strategy
function onWithdraw(uint256 _amountToWithdraw)
    external override(IStrategyTemplate) whenNotPaused onlyRoycoVault
    returns (uint256 amountWithdrawn)
{
    // ...
    amountWithdrawn = IMachine(MAKINA_MACHINE).redeem(...);
}

// Machine
function redeem(uint256 shares, address receiver, uint256 minAssets)
    external override nonReentrant notRecoveryMode returns (uint256)
{
    // ...
}
```

When either condition is active, `maxWithdraw()` returns a positive value but `onWithdraw(...)` reverts. Because the Concrete vault's `_executeWithdraw(...)` loop has no try-catch around the `onWithdraw(...)` call, the revert propagates and fails the entire withdrawal transaction. The vault cannot fall back to the next strategy in `deallocationOrder`, blocking all user withdrawals that would need to pull from this strategy, even if subsequent strategies had sufficient liquidity.

Recommendation(s): Consider returning `0` from `maxWithdraw()` when the strategy is paused or when the Machine is in recovery mode.

Status: Fixed

Update from the client: Fixed in both `maxWithdraw(...)` and `maxAllocation(...)`.

6.6 [Best Practices] Undocumented calldata structure for allocation and deallocation parameters

File(s): src/RoycoVaultMakinaStrategy.sol

Description: The `allocateFunds(...)` and `deallocateFunds(...)` functions expect a `64`-byte `bytes calldata` parameter that is decoded using inline assembly. However, the expected structure of this calldata (two `uint256` values packed sequentially) is not documented in the function's NatSpec or elsewhere in the contract. This forces callers to inspect the assembly implementation to understand how to properly encode parameters. This is an example of internal implementation details leaking to external callers, which is considered poor API design.

Recommendation(s): Consider adding NatSpec documentation to both functions describing the expected calldata format, specifying that the first `32` bytes should contain the amount (or shares) and the second `32` bytes should contain the minimum output value.

Status: Fixed

Update from the client: Fixed. Added NatSpec comments.

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 Royco Protocol's contracts documentation

The `Royco Protocol's` team provided an overview of the system components during the kick-off call with a detailed explanation of the intended functionalities and use cases of the system and a written README file that explained the codebase. Moreover, the 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

The Royco protocol's test suite is robust and clearly outlines the contract specifications. The test suite catches any change to the business logic. To assess the test coverage for these files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. 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 to "kill" the mutant.

The following table outlines the results of the analysis performed on the Royco's core smart 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. The higher the score, the better the test suite is at finding bugs.

Contract	Mutants (slain / total generated)	Score
`RoycoVaultMakinaStrategy.sol`	44 / 45	97.78%
Total	44 / 45	97.78%

Remarks about the Royco Protocol Test Suite

The test suite for the Royco Protocol demonstrates excellent coverage of core business logic with an overall mutation score of 97.78%, which is well above the average. This confirms that the strategy's business logic is thoroughly tested.

The only undertested area, identified by a single unslain mutant, relates to the adjustment of shares to redeem in the `onWithdraw(...)` function. Notably, this is precisely the area where the sole Low severity finding was discovered during the code review phase. Mutation testing can be incorporated into the development process to catch similar bugs in the future.

It is recommended to add more test cases for scenarios where the asynchronous vault could be turned off and would become synchronous, ensuring the strategy handles such state transitions correctly.

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.

RoycoVaultMakinaStrategy.sol

In the `onWithdraw(...)` function, the line `sharesToRedeem = Math.min(sharesToRedeem, _getStrategyOwnedShares())` can be replaced with `assert(true)` (which is effectively a no-op) without affecting the test suite. This `Math.min` is intended to cap the shares to redeem when the `+1` rounding padding would exceed the strategy's owned shares. The test suite should include:

- A test case where the user attempts to withdraw slightly less than their full position, such that `convertToShares(withdrawAmount) + 1` exceeds `_getStrategyOwnedShares()`, verifying that the function correctly caps the shares and does not revert.
- A boundary test where the Makina machine has limited liquidity (e.g., funds deployed to Calibers), and the withdrawal amount equals exactly `machine.maxWithdraw()`. This scenario triggers the Low severity finding where the `+1` padding causes the redemption to exceed available liquidity.
- An integration test simulating the Royco vault's `_executeWithdraw` loop calling `onWithdraw` with amounts derived from `strategy.maxWithdraw()`, verifying the strategy handles the machine liquidity boundary correctly.

8.1 Automated Tools

8.1.1 AuditAgent

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

9 About Nethermind

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

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

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

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

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

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

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

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

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

General Advisory to Clients

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

Disclaimer

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