Security Review Report
NM-0442-Token-Fleet

NETHERMIND
SECURITY

(April 7, 2025)

Contents

1 Executive Summary **2**

2 Audited Files **3**

3 Summary of Issues **3**

4 System Overview **4**
4.1 Admin Flow . . . 4
4.2 Users Flow . . . 5

5 Risk Rating Methodology **6**

6 Issues **7**
6.1 [High] `Deal::cancel` uses `totalSupply` instead of `totalAssets` to verify fundraising target . . . 7
6.2 [Medium] Inconsistent update of `lastUpdate` . . . 8
6.3 [Medium] Referral tokens can be permanently locked for portfolio owners without referrers . . . 9
6.4 [Low] `Deal::closeDeal` does not check `dealDealine` . . . 10
6.5 [Low] `Portfolio` contract does not expose approve function for asset transfers to `Deal` contract . . . 11

7 Documentation Evaluation **12**

8 Test Suite Evaluation **13**
8.1 Compilation Output . . . 13
8.2 Tests Output . . . 13
8.3 Automated Tools . . . 14
8.3.1 AuditAgent . . . 14

9 About Nethermind **15**

1 Executive Summary

This document presents the security review performed by Nethermind Security for the smart contracts of Token Fleet. Token Fleet is a blockchain-powered platform that allows users to invest in cars through fractional ownership.

By owning tokens, users gain access to a diverse fleet of cars, earning a share of the rental profits and eventual sale proceeds. This allows users to tap into the car rental market without the complexities of full ownership.

This security review focuses exclusively on the smart contracts listed in Section 2 (*Audited Files*). **The audited code comprises of** 769 lines of code written in the Solidity language, and the audit was performed using (a) manual analysis of the code base and (b) creation of test cases. **Along this document, we report** 5 points of attention, where one is classified as `High`, two are classified as `Medium`, and two are classified as `Low`. 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 overview of the system. 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.

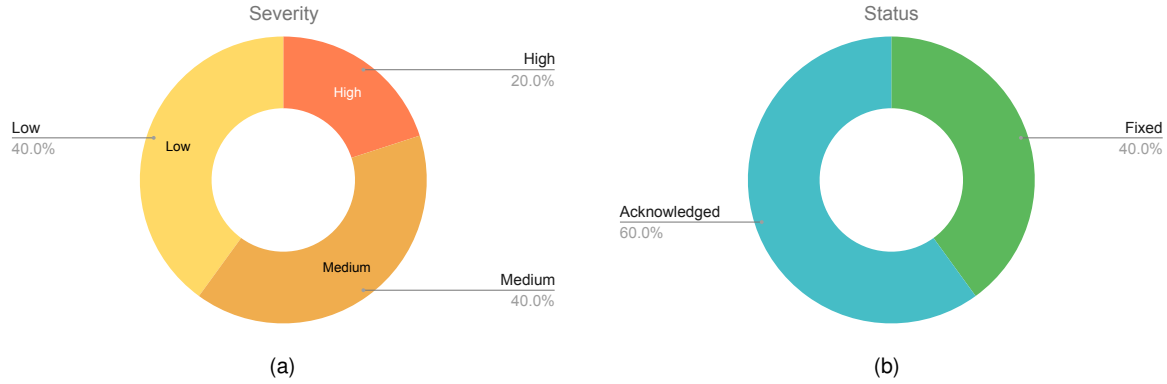

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (2), **Low** (2), **Undetermined** (0), **Informational** (0), **Best Practices** (0). **Distribution of status: Fixed** (2), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Response from Client	Regular responses during audit engagement
Final Report	April 7, 2025
Repository	Token Fleet
Commit (Audit)	c036ac461477624bdcb1fdbfddeb64e66d6a8cda
Commit (Final)	4d82b7891d6c5f819ac20ac3921af183af83e162
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	PortfolioRegistry.sol	161	24	14.9%	36	221
2	Deal.sol	352	94	26.7%	104	550
3	InsuranceVault.sol	7	1	14.3%	2	10
4	Portfolio.sol	81	24	29.6%	21	126
5	DealManager.sol	168	37	22.0%	43	248
	Total	**769**	**180**	**23.4%**	**206**	**1155**

3 Summary of Issues

	Finding	Severity	Update
1	`Deal::cancel` uses `totalSupply` instead of `totalAssets` to verify fundraising target	High	Fixed
2	Inconsistent update of `lastUpdate`	Medium	Fixed
3	Referral tokens can be permanently locked for protfolio owners without referrers	Medium	Acknowledged
4	`Deal::closeDeal` does not check `dealDealine`	Low	Acknowledged
5	`Portfolio` contract does not expose approve function for asset transfers to `Deal` contract	Low	Acknowledged

4 System Overview

4.1 Admin Flow

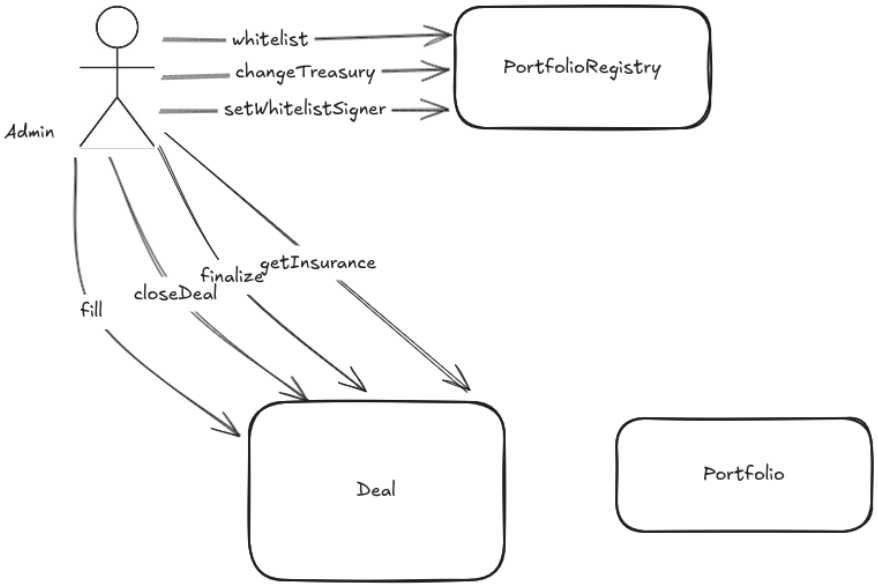

Fig. 2: Admin flow overview

The Admin Flow is designed for administrators who oversee the creation and management of Deals.

- Admins initiate the process by creating a Deal through the `createDeal` function in the `DealManager` contract. Each deal has specific parameters, such as a fundraising target, deadlines, fees, and insurance requirements.
- Admins can cancel a deal if the fundraising target is not met within the specified deadline by calling `cancelDeal`.
- Once the target is reached, the admin can close the deal using `closeDeal`, transferring the raised funds to the designated dealer and allocating insurance funds to a vault.
- Throughout the deal's lifecycle, admins can add income generated by the deal using `addIncome`, which distributes profits to investors after deducting fees. If additional insurance funds are required, admins can withdraw them using `getInsurance`.
- Finally, when all assets associated with the deal are sold, the admin can finalize the Deal using `finalizeDeal`, ensuring all remaining funds are distributed appropriately.

Admins also have the ability to upgrade contract implementations at any time.

4.2 Users Flow

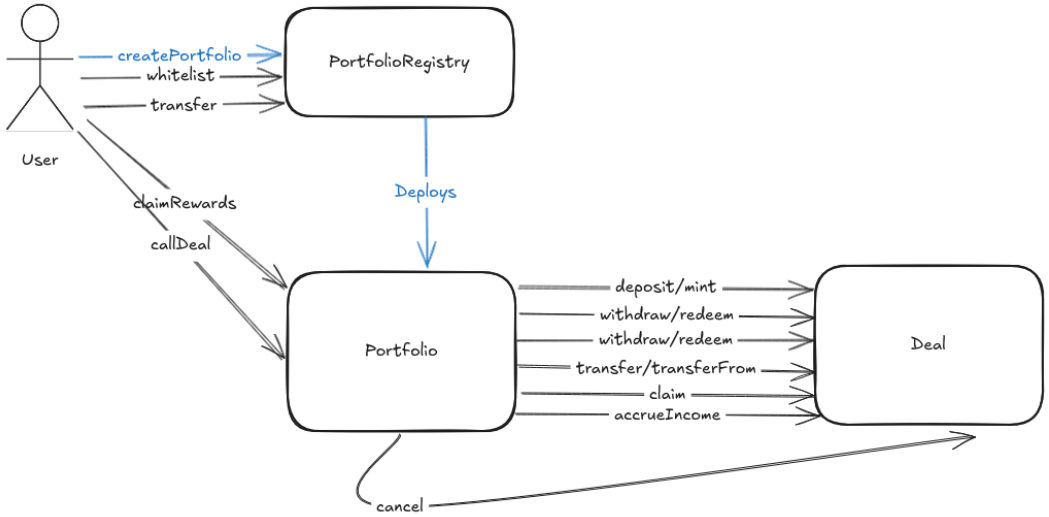

Fig. 3: User flow overview

The User Flow is tailored for investors who interact with deals through their portfolios.

- Users begin by creating a portfolio via the `createPortfolio` function in the `PortfolioRegistry` contract. Each portfolio is represented as an ERC721 token, allowing users to manage multiple portfolios if needed.
- Users can interact with deals through their portfolios by calling `callDeal`, which enables them to invoke any function on a deal, such as depositing funds, claiming rewards, or canceling their participation.
- All token transfers within the system are restricted to portfolios, ensuring that funds remain within the ecosystem. Users can also specify a `rewardReceiver` address to receive rewards generated by their interactions with deals.
- Additionally, users can directly interact with deals for functions that do not involve token transfers, such as canceling their participation, accruing income, or depositing assets.
- Rewards accumulated in a portfolio can be claimed using `claimRewards`, and these rewards can also be used for future deposits

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] `Deal::cancel` uses `totalSupply` instead of `totalAssets` to verify fundraising target

File(s): `Deal.sol`

Description: Each deal has two deadlines: (a) a fundraising deadline; and, (b) a closing deadline. Users can cancel a deal if either deadline is not met.

```
function cancel() external {
    DealStorage storage $ = _getDealStorage();
    _onlyStatus($, STATUS.RAISING_FUNDS);

    if (totalSupply() < $.target) {
        if (block.timestamp <= $.fundraisingDeadline) {
            revert FundraisingDeadlineNotReached();
        }
    } else {
        if (block.timestamp <= $.dealDeadline) {
            revert DealDeadlineNotReached();
        }
    }
    $.status = STATUS.CANCELLED;
    emit DealCancelled();
}
```

The issue arises in how the target is checked. It incorrectly relies on `totalSupply` instead of `totalAssets`. Given that the ratio between the asset and the supply token is not 1:1 and instead linear (for example, a malicious user can intentionally fund the Deal contract with the assets directly through a transfer), the usage of `totalSupply` is incorrect.

Recommendation(s): Modify the mentioned check to use `totalAssets()` instead of `totalSupply()`:

```
function cancel() external {
    DealStorage storage $ = _getDealStorage();
    _onlyStatus($, STATUS.RAISING_FUNDS);

-   if (totalSupply() < $.target) {
+   if (totalAssets() < $.target) {
        if (block.timestamp <= $.fundraisingDeadline) {
            revert FundraisingDeadlineNotReached();
        }
    } else {
        if (block.timestamp <= $.dealDeadline) {
            revert DealDeadlineNotReached();
        }
    }
    $.status = STATUS.CANCELLED;
    emit DealCancelled();
}
```

Status: Fixed.

Update from the client: Fixed in commit 2d447e0

6.2 [Medium] Inconsistent update of `lastUpdate`

File(s): `Deal.sol`

Description: Users who deposit funds into deals are entitled to proportional rewards. They claim these rewards by calling `Deal::claim`, which updates `lastUpdate[user]` to ensure the claimed portion is excluded from future claims:

```
function _claim(DealStorage storage $, address user, address receiver) internal returns (uint256) {
    uint256 assets = reward(user);

==>     $.lastUpdate[user] = $.totalIncome;

    if (assets > 0) {
        USDC.safeTransfer(receiver, assets);
    }

    emit Claimed(user, receiver, assets);
    return assets;
}

function reward(address user) public view returns (uint256) {
    DealStorage storage $ = _getDealStorage();
    uint256 shares = balanceOf(user);
    return Math.mulDiv(shares, $.totalIncome - $.lastUpdate[user], totalSupply(), Math.Rounding.Floor);
}
```

However, a rounding issue can cause the claimable assets to be zero, despite updating `lastUpdate[user]`. This results in the system marking the user as having claimed their income when they actually received nothing.

Consider the following example:

- The total assets deposited into the deal are `5M` USDC, and the total supply is approximately `3M` shares;
- Bob deposits `1.6K` USDC, receiving `1K` shares;
- `2.5K` USDC is added to the deal contract as income;
- Bob calls `claim` to receive his share of the income. The reward function computes: `1K×2.5K / 3M = 0`;
- Since `assets` is `0`, Bob receives no rewards, but `lastUpdate[user]` is still updated, preventing him from claiming his rightful share in future distributions;

Recommendation(s): Update `lastUpdate[user]` only when the calculated rewards are greater than zero:

```
function _claim(DealStorage storage $, address user, address receiver) internal returns (uint256) {
    uint256 assets = reward(user);

-   $.lastUpdate[user] = $.totalIncome;

    if (assets > 0) {
+       $.lastUpdate[user] = $.totalIncome;
        USDC.safeTransfer(receiver, assets);
    }

    emit Claimed(user, receiver, assets);
    return assets;
}
```

Status: Fixed.

Update from the client: Fixed in commits 4b322a8 and 4d82b78

6.3 [Medium] Referral tokens can be permanently locked for portfolio owners without referrers

File(s): DealManager.sol

Description: When raising funds for a deal via `Deal::deposit`, the `DealManager::registerDeposit` function is invoked. This function tracks the deposited amount for each deal to account for referral rewards:

```
function registerDeposit(uint256 assets, address receiver) external {
    DealManagerStorage storage $ = _getDealManagerStorage();

    PortfolioRegistry registry = PortfolioRegistry($.portfolioRegistry);
    address owner = registry.onlyWhitelistedPortfolio(uint256(uint160(receiver)));
==>    address referrer = registry.referrer(owner);
    $.deals[msg.sender].deposits += assets;
==>    $.deals[msg.sender].referralDeposits[referrer] += assets;
}
```

Later, users can call `claimReferralRewards` to claim their referral rewards, which are distributed proportionally:

```
function _referralRewards(DealManagerStorage storage $, address deal, address user)
    internal
    view
    returns (uint256)
{
    DealData storage dealData = $.deals[deal];
    return Math.mulDiv(
        dealData.referralDeposits[user],
        dealData.rewards - dealData.lastRewardClaim[user],
        dealData.deposits,
        Math.Rounding.Floor
    );
}
```

However, some portfolio owners may not have a referrer, causing `registry.referrer(owner);` to return the zero address. In such cases, referral rewards are permanently locked to the zero address.

Recommendation(s): Modify `registerDeposit` to update `referralDeposits` only when a valid referrer exists:

```
function registerDeposit(uint256 assets, address receiver) external {
    DealManagerStorage storage $ = _getDealManagerStorage();

    PortfolioRegistry registry = PortfolioRegistry($.portfolioRegistry);
    address owner = registry.onlyWhitelistedPortfolio(uint256(uint160(receiver)));
    address referrer = registry.referrer(owner);
    $.deals[msg.sender].deposits += assets;
-   $.deals[msg.sender].referralDeposits[referrer] += assets;
+   if (referrer != address(0)) $.deals[msg.sender].referralDeposits[referrer] += assets;
}
```

Status: Acknowledged.

Update from the client: At the moment, all users need to have a referrer when they get whitelisted, but there is no check enforcing that the referrer is not 0x0. However, only the owner or the `whitelistSigner` can make that mistake, and we treat them as trustworthy.

6.4 [Low] `Deal::closeDeal` does not check `dealDealine`

File(s): `Deal.sol`

Description: Each deal has two deadlines: (a) a fundraising deadline and (b) a closing deadline. Users can cancel a deal if either deadline is not met. After successful fundraising, the deal manager calls `Deal::closeDeal` to transition the deal to the next phase. However, the function does not check whether the `dealDeadline` has already passed:

```
function closeDeal(address dealer) external {
    DealStorage storage $ = _getDealStorage();
    _onlyDealManagerOwner($, _msgSender());
    _onlyStatus($, STATUS.RAISING_FUNDS);
    _nonZero(dealer);

    if (totalAssets() < $.target) {
        revert TargetNotMet();
    }

    uint256 treasuryFee = $.treasuryFee;
    uint256 insuranceAmount = $.insurance;
    uint256 dealAssets = totalAssets() - insuranceAmount - treasuryFee;

    // Start deal
    $.status = STATUS.IN_PROGRESS;

    USDC.approve(address($.dealManager), treasuryFee);
    $.dealManager.processFee(treasuryFee);
    USDC.safeTransfer(dealer, dealAssets);
    USDC.safeTransfer(address($.insuranceVault), $.insurance);

    emit DealClosed(dealer);
}
```

While the `cancelDeal` function allows users to cancel an expired deal and we expect them to cancel deals closely after their deadline, the `closeDeal` function should still validate the `dealDeadline` to prevent unintended deal progression past the allowed timeframe.

Recommendation(s): Add a check in `closeDeal` to check for the deal's deadline

Status: Acknowledged.

Update from the client: Deadlines are meant as a mechanism for users to recover their assets if there are delays in executions. Given that, if there are delays and no one cancels, we are fine with closing the deal anyway, as it just means users did not want to cancel. Deadline is a safety mechanism for the user, not for the deal breaker for our processes.

6.5 [Low] `Portfolio` contract does not expose approve function for asset transfers to `Deal` contract

File(s): `Portfolio.sol`

Description: Users are meant to interact with deal contracts through their portfolios. For instance, portfolios can interact with the `Deal` contract to deposit the underlying assets. However, the issue is that depositing/minting assets on the `Deal` contract requires transfer approval of the asset, and the `Portfolio` contract does not expose an approve function to grant spending allowances to the `Deal` contract, which leads to transactions reverts. The current implementation of `callDeal` allows a workaround of this as it performs low-level calls to any address and not necessarily the `Deal` contract, in which the portfolio owners can leverage this function to interact with the underlying asset to approve for the deal contract.

```
function _call(address deal, bytes calldata data, address rewardsReceiver) internal returns (bytes memory) {
    (bool success, bytes memory result) = address(deal).call(data);
    if (!success) {
        assembly {
            revert(add(0x20, mload(0)), mload(0))
        }
    }
    _claimRewards(rewardsReceiver);
    return result;
}
```

However, this is not the intended behavior of the function.

Recommendation(s): Consider exposing a function to approve a token to be spent by a deal address

Status: Acknowledged.

Update from the client: It may be a nice-to-have, but a workaround exists using callDeal, and we may stick with this approach.

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 provide a reference for developers who need to modify or maintain it in the future.

Remarks about Token Fleet documentation

The `Token Fleet` team was actively present in regular calls, effectively addressing concerns and questions raised by the Nethermind Security team. The codebase included natspec and protocol description that were insightful for the Nethermind Security team to understand the codebase.

8 Test Suite Evaluation

8.1 Compilation Output

```
$ forge compile
[] Compiling...
[] Compiling 83 files with Solc 0.8.26
[] Solc 0.8.26 finished in 3.67s
Compiler run successful!
```

8.2 Tests Output

```
$ forge test
Ran 7 tests for test/DealManager.t.sol:DealManagerTest
[PASS] testCreateDeal(string,uint256,uint256,uint256,uint256,uint256,uint256) (runs: 256, : 529151, ~: 562465)
[PASS] testDealCannotBeCreatedTwice(string,uint256,uint256,uint256,uint256,uint256,uint256) (runs: 256, : 1072687535,
↪  ~: 1072687519)
[PASS] testInitialization() (gas: 149577)
[PASS] testOnlyOwnerCanCreateDeal(string,uint256,uint256,uint256,uint256,uint256,uint256,address) (runs: 256, : 13184,
↪  ~: 13181)
[PASS] testOnlyOwnerCanSetTreasury(address,address) (runs: 256, : 12046, ~: 12046)
[PASS] testSetTreasury(address) (runs: 256, : 19257, ~: 19257)
[PASS] testTreasuryCannotBeZero() (gas: 10984)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 114.62ms (208.26ms CPU time)

Ran 4 tests for test/PortfolioRegistry.t.sol:PortfolioRegistryInitializationTest
[PASS] testInitialization(address,address) (runs: 256, : 301302, ~: 301302)
[PASS] testWrongBeaconAddress(address,address,address) (runs: 256, : 4140944, ~: 140804)
[PASS] testZeroOwner(address) (runs: 256, : 80385, ~: 80385)
[PASS] testZeroTreasury(address) (runs: 256, : 247506, ~: 247506)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 382.80ms (76.45ms CPU time)

Ran 5 tests for test/Deal.t.sol:DealTestInitialization
[PASS] testInitializeFailsIfFeeIsGreaterThan100(string,uint256,uint256,uint256,uint256,address,uint256) (runs: 256, :
↪  329034, ~: 398246)
[PASS] testInitializeFailsIfFeeIsZero(string,uint256,uint256,uint256,address,uint256) (runs: 256, : 312950, ~: 286919)
[PASS] testInitializeFailsIfInsuranceIsNotIncluededInTheTarget(string,uint256,uint256,uint256,uint256,uint256,address,
uint256) (runs: 256, : 175002, ~: 208905)
[PASS] testInitializeFailsIfManagerIsZeroAddress(string,uint256,uint256,uint256,uint256,uint256,uint256) (runs: 256, :
↪  166224, ~: 163722)
[PASS] testValidInitialization(string,address,uint256,uint256,uint256,uint256,uint256,uint256) (runs: 256, : 470744, ~:
↪  504842)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 625.14ms (192.39ms CPU time)

Ran 8 tests for test/PortfolioRegistry.t.sol:PortfolioRegistryTest
[PASS] testChangeTreasury(address) (runs: 256, : 25534, ~: 25534)
[PASS] testCheckPortfolioOwner() (gas: 215005)
[PASS] testNewTreasuryIsNotZero() (gas: 13356)
[PASS] testOnlyOwnerCanChangeTreasury(address) (runs: 256, : 15736, ~: 15736)
[PASS] testOwnerWhitelist(address) (runs: 256, : 38418, ~: 38418)
[PASS] testUserWhitelist(address,uint256,uint256,address) (runs: 256, : 291217, ~: 291313)
[PASS] testWhitelistInvalidSignature() (gas: 19358)
[PASS] testWhitelistNonceAlreadyUsed(uint256) (runs: 256, : 58459, ~: 58459)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 625.26ms (600.63ms CPU time)
```

```
Ran 43 tests for test/Deal.t.sol:DealTest
[PASS] testAccrueIncomeFailsWhenNotInProgress(uint256) (runs: 256, : 212996, ~: 213004)
[PASS] testAccrueIncomeSuccessful(uint256) (runs: 256, : 674950, ~: 674817)
[PASS] testAccrueIncomeWithInsurance(uint256,uint256) (runs: 256, : 840078, ~: 841943)
[PASS] testCancelFailsIfDealDeadlineNotReached(uint256) (runs: 256, : 329047, ~: 329085)
[PASS] testCancelFailsIfFundraisingDeadlineNotReached(uint256,uint256) (runs: 256, : 330048, ~: 329848)
[PASS] testCancelFailsIfNotRaising(uint8) (runs: 256, : 28099, ~: 28156)
[PASS] testCancelIfDealDeadlineIsPassed(uint256) (runs: 256, : 333475, ~: 333480)
[PASS] testCancelIfFundraisingDeadlineIsPassed(uint256,uint256) (runs: 256, : 334431, ~: 334513)
[PASS] testClaimRoundingErrors() (gas: 1283921)
[PASS] testClaimSuccessful(uint256) (runs: 256, : 751503, ~: 751631)
[PASS] testCloseDealFailsWhenAlreadyClosed(address) (runs: 256, : 482198, ~: 482181)
[PASS] testCloseDealFailsWhenNotDealManager(address,address) (runs: 256, : 343836, ~: 343804)
[PASS] testCloseDealFailsWhenTargetNotMet(uint256,address) (runs: 256, : 345521, ~: 345552)
[PASS] testCloseDealFailsWithInvalidDealer() (gas: 340984)
[PASS] testCloseDealSuccessful(address) (runs: 256, : 497987, ~: 497970)
[PASS] testCloseDealWithExactTarget(address) (runs: 256, : 482960, ~: 482943)
[PASS] testDepositEmitsTransferEvent() (gas: 336400)
[PASS] testDepositFailsWhenAmountIsLessThanMinInvestment(uint256) (runs: 256, : 23382, ~: 23695)
[PASS] testDepositFailsWhenExceedingTarget() (gas: 221086)
[PASS] testDepositFailsWhenNotRaisingFunds() (gas: 493266)
[PASS] testDepositSuccessful(uint256) (runs: 256, : 346789, ~: 346828)
[PASS] testDepositUpToTarget() (gas: 344076)
[PASS] testDepositWithDifferentReceiver() (gas: 449631)
[PASS] testFinalizeFailsWhenDealNotInProgress() (gas: 29401)
[PASS] testFinalizeFailsWhenNotDealManager(address) (runs: 256, : 499867, ~: 499850)
[PASS] testFinalizeSuccessful() (gas: 685611)
[PASS] testGetInsuranceFailsWhenNotDealManager(address) (runs: 256, : 30787, ~: 30787)
[PASS] testGetInsuranceSuccessful(uint256) (runs: 256, : 604844, ~: 605036)
[PASS] testGetInsuranceWithExcessiveAmount() (gas: 572310)
[PASS] testMintFailsWhenAmountIsLessThanMinInvestment(uint256) (runs: 256, : 34047, ~: 34350)
[PASS] testMintFailsWhenExceedingTarget() (gas: 222844)
[PASS] testMintFailsWhenNotRaisingFunds() (gas: 494815)
[PASS] testMintSuccessful(uint256) (runs: 256, : 364847, ~: 364893)
[PASS] testMultipleClaimsAndAccruals(uint256,uint256) (runs: 256, : 1567895, ~: 1567542)
[PASS] testMultipleDeposits(address,address,uint256,uint256) (runs: 256, : 856802, ~: 856949)
[PASS] testOnlyOwnerCanFill(uint256,address) (runs: 256, : 717785, ~: 717808)
[PASS] testReferralRewards(address,address) (runs: 256, : 1554322, ~: 1554351)
[PASS] testReinitializationFail(string,uint256,uint256,uint256,address,uint256,uint256,uint256) (runs: 256, : 33566, ~:
↪  33359)
[PASS] testRewardAfterAWhileWithoutShares(uint256,uint256,uint256) (runs: 256, : 1734996, ~: 1738410)
[PASS] testRewardReturn0AfterBurningAndGettingBackShares(uint256,uint256,uint256) (runs: 256, : 1430668, ~: 1434223)
[PASS] testRewardShouldReturnTheSameForOthersAfterBurningShares(uint256,uint256,uint256) (runs: 256, : 1335676, ~:
↪  1339822)
[PASS] testSharesEqualAssetsAtEndOfRaising() (gas: 344175)
[PASS] testTransferAutomaticallyClaims() (gas: 965578)
Suite result: ok. 43 passed; 0 failed; 0 skipped; finished in 625.31ms (6.64s CPU time)

Ran 5 test suites in 626.46ms (2.37s CPU time): 67 tests passed, 0 failed, 0 skipped (67 total tests)
```

8.3 Automated Tools

8.3.1 AuditAgent

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

9 About Nethermind

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

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

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

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

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

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

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

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

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

Learn more about us at nethermind.io.

General Advisory to Clients

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

Disclaimer

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