Security Review Report
NM-0648 - TruStake POL

NETHERMIND SECURITY

(September 19, 2025)

Contents

1 Executive Summary

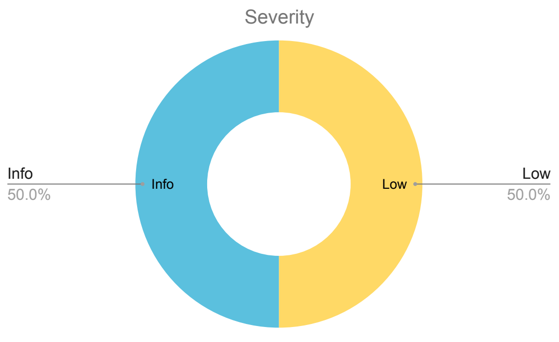
This document presents the results of a security review conducted by Nethermind Security for Trufin's `TruStakePOL` contract.

The **TruStakePOL** contract resembles an ERC4626 vault contract, but is heavily customized. The contract enables users to deposit POL tokens in exchange for TruPOL vault shares. The POL tokens are then staked by the vault with a Polygon chain validator. The contract keeps compounding the rewards for its users, generating yield from network staking. The contract is deployed behind a transparent proxy, making it upgradeable.

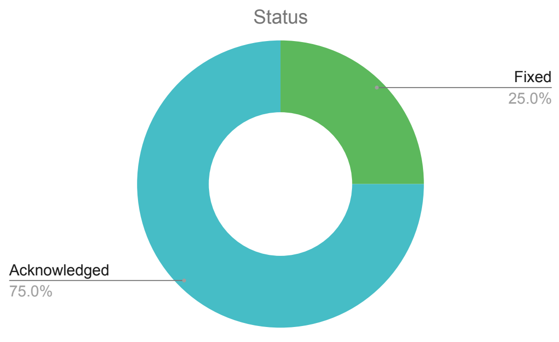
The audit comprises 549 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 4 points of attention where two are classified as `Low` and two are classified as `Informational` severity. 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 test suite evaluation and automated tools used. Section 9 concludes the document.



(a)



(b)

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

Summary of the Audit

Audit Type	Security Review
Initial Report	September 12, 2025
Final Report	September 19, 2025
Initial Commit	71092263f4740669e28c170095e42d0f8a70d644
Final Commit	ba7f1a7ede4668f15132ab08a7f4baa9a8a33bca
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/pol-staker/contracts/main/TruStakePOLStorage.sol	18	17	94.4%	3	38
2	contracts/pol-staker/contracts/main/TruStakePOL.sol	505	161	31.9%	133	799
3	contracts/pol-staker/contracts/main/Types.sol	26	12	46.2%	5	43
	Total	**549**	**190**	**34.6%**	**141**	**880**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect rounding favors withdrawing users	Low	Fixed
2	No mechanism to reflect slashing events	Low	Acknowledged
3	Unbounded loop over validators may lead to denial of service	Info	Acknowledged
4	`maxWithdraw` calculation ignores per-validator liquidity constraints	Info	Acknowledged

4 Protocol Overview

The TruStake POL vault provides access to POL staking on the Ethereum network. When you deposit in the vault, your POL is immediately sent to one institutional validator, and rewards start to accrue immediately. Rewards are restaked automatically, uplifting APY from the compounding effect.

4.1 Staking

The staking process is initiated by depositing POL into the vault, resulting in minting an equivalent amount of TruPOL as shares back to the user. The deposited POL is directly staked within the validator. Deposits are possible through the `deposit(uint256 _assets)` and `depositToSpecificValidator(uint256 _assets, address _validator)` functions, where the former uses the default validator set in the contract and the latter allows the user to designate one of the supported validators.

4.2 Unstaking

The unstaking process involves two phases:

- The user initiates a withdrawal request for a specific amount of POL, resulting in the vault burning the equivalent TruPOL shares and triggering an unbonding process on the validator. Similar to deposits, users can withdraw from a designated validator or use TruStake's default validator, using `withdraw(uint256 _assets)` and `withdrawFromSpecificValidator(uint256 _assets, address _validator)` functions respectively.
- Upon completion of the withdrawal delay, users can claim their funds via the `withdrawClaim(uint256 _unbondNonce, address _validator)` function by providing the unbonding nonce that is generated in the request phase.

4.3 Auto-compounding

The vault provides the `compoundRewards(...)` function that is periodically called to restake rewards across different validators and stake any claimed rewards and available POL in the vault.

4.4 Fees

The TruStake vault takes a fee on the earned rewards. This fee is paid to the treasury in the form of TruStake shares.

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] Incorrect rounding favors withdrawing users

File(s): `contracts/pol-staker/contracts/main/TruStakePOL.sol`

Description: When a user withdraws a specific amount of assets (`_amount`) from the vault, the `_withdrawRequest(...)` function calculates the corresponding number of shares to burn. This calculation uses standard integer division, which implicitly rounds the result down.

```
function _withdrawRequest(address _user, uint256 _amount, address _validator)
    private
    returns (uint256 shareDecreaseUser, uint256 unbondNonce)
{
    // ...
            } else {
                // calculate share decrease
                // @audit-issue The result of this division is rounded down.
                shareDecreaseUser = (_amount * globalPriceDenom * WAD) / globalPriceNum;
            }
    // ...
    _burn(_user, shareDecreaseUser);
    // ...
}
```

By rounding down, the contract burns slightly fewer shares than are required to cover the withdrawn assets. This rounding error consistently benefits the withdrawing user at the expense of all remaining liquidity providers in the vault. Over a large number of withdrawal transactions, these small discrepancies can accumulate, leading to a gradual drain of value from the protocol.

An evil user could inflate the share price to make the rounding error bigger and execute multiple withdrawals without spending shares, effectively draining par of the assets of the vault.

Recommendation(s): Consider changing the rounding direction when calculating shares to burn during a withdrawal. The calculation should round *up* to ensure that the user provides a sufficient amount of shares to cover the assets they are taking.

Status: Fixed.

Update from the client: This has been fixed in branch pol-fix-epsilon, where we remove the epsilon value that was making it possible for the contract to favor users during withdrawals.

6.2 [Low] No mechanism to reflect slashing events

File(s): `contracts/pol-staker/contracts/main/TruStakePOL.sol`

Description: The `TruStakePOL` contract tracks the amount of POL staked with each validator using the `stakedAmount` field within the `Validator` struct. This value is a critical component of the `totalStaked()` calculation, which in turn determines the vault's share price.

The `stakedAmount` is updated only through internal actions such as staking (`_stake`), unbonding (`_unbond`), and restaking rewards (`_restake`). The protocol lacks a mechanism to verify if this internally tracked amount matches the actual balance of staked POL held by the external `ValidatorShare` contracts.

```
struct Validator {
    ValidatorState state;
    // @audit-issue This value is trusted internally but not verified against the external reality.
    uint256 stakedAmount;
    address validatorAddress;
}
```

If a validator is slashed, its actual staked POL balance will decrease. However, the `stakedAmount` variable in the `TruStakePOL` contract will not be updated to reflect this loss.

This discrepancy leads to an overestimation of the vault's total assets. The `totalStaked()` function will return an inflated value, causing the `sharePrice()` to be calculated incorrectly as higher than it should be. The vault will become undercollateralized, where the value of the issued shares exceeds the value of the underlying assets. This situation benefits savvy users who withdraw early, as they can redeem their shares for more POL than they are worth, at the expense of all other users. The last users to withdraw may find that there are insufficient assets left to cover their shares, resulting in a loss of funds.

Additionally, calls to `buyVoucherPOL(...)` and `sellVoucher_newPOL(...)` require the share price from the `ValidatorShare` to be at least `one`, which may not be the case in case of slashing, affecting deposit and withdrawals mechanisms.

Recommendation(s): Consider implementing a mechanism to account for slashing events.

Status: Acknowledged.

Update from the client: Slashing is currently **disabled on Polygon**, with no confirmed timeline for activation. While there is some preliminary code in the validator contracts to support slashing, its final implementation, parameters, and enforcement mechanisms remain undefined. As such, this issue is not currently actionable.

6.3 [Info] Unbounded loop over validators may lead to denial of service

File(s): contracts/pol-staker/contracts/main/TruStakePOL.sol

Description: The contract owner has the ability to add new validators to the system via the addValidator(...) function, which appends the new validator's address to the _validatorAddresses storage array. While validators can be disabled using disableValidator(...), there is no mechanism to remove them from this array.

Several core functions, including totalStaked(), totalRewards(), and _restake() (called by compoundRewards(...)), iterate over the entire _validatorAddresses array to perform their calculations.

```
function totalStaked() public view override returns (uint256) {
    TruStakePOLStorageStruct storage $ = _getTruStakePOLStorage();
    uint256 validatorCount = $._validatorAddresses.length;
    uint256 stake;
    // @audit-issue This loop's cost grows with the number of validators.
    for (uint256 i; i < validatorCount; ++i) {
        stake += $._validators[$._validatorAddresses[i]].stakedAmount;
    }
    return stake;
}
```

The issue is that the _validatorAddresses array can grow indefinitely. As more validators are added over time, the gas cost of executing these functions will increase linearly. Eventually, the transaction cost could exceed the block gas limit, rendering these critical functions unusable.

This would create a permanent Denial of Service (DoS), with the following impacts:

- compoundRewards(...) would become uncallable, preventing all rewards from being restaked.
- totalStaked() and totalRewards() would revert, breaking the sharePrice() calculation and thus all deposit and withdrawal operations.

Recommendation(s): Consider introducing a mechanism to remove validators from the _validatorAddresses array. When a validator is no longer needed, it should be fully decommissioned and removed.

Status: Acknowledged.

Update from the client: While we acknowledge the concern, this loop is not expected to pose a practical risk in our specific context. The Polygon validator set is capped at 105 validators, and at TruFin, we operate with a tightly controlled validator onboarding process. All validators are KYC'd and whitelisted, and we expect to onboard only a small, static subset of the available set. As such, the number of active validator entries will remain low and predictable, ensuring the gas cost of looping operations remains well within safe bounds.

6.4 [Info] `maxWithdraw` calculation ignores per-validator liquidity constraints

File(s): `contracts/pol-staker/contracts/main/TruStakePOL.sol`

Description: The `maxWithdraw(...)` function is a view function usually intended to inform users of the maximum amount of POL they can withdraw in one call. The withdrawal logic, handled by `_withdrawRequest(...)`, correctly enforces that a withdrawal from a specific validator cannot exceed the amount staked with that validator.

However, the `maxWithdraw(...)` function calculates the withdrawable amount based on the user's total share holdings, without considering how the underlying liquidity is distributed across the various validators.

```
function maxWithdraw(address _user) public view override returns (uint256) {
    // @audit This calculation is based on the user's total shares.
    uint256 preview = previewRedeem(balanceOf(_user));
    if (preview == 0) return 0;

    TruStakePOLStorageStruct storage $ = _getTruStakePOLStorage();

    // @audit-issue It does not account for the stake on any single validator.
    return preview + $._epsilon;
}
```

This creates a discrepancy that can affect user's experience. A user might check `maxWithdraw(...)`, see a large available amount, and attempt to withdraw it using the `withdraw(...)` function, which targets the default validator. If the default validator's staked amount is less than the user's requested amount, the transaction will revert with a `WithdrawalAmountAboveValidatorStake` error, even though the user was led to believe the withdrawal was possible. This can cause user confusion and wasted gas on failed transactions.

Recommendation(s): Consider modifying the `maxWithdraw(...)` function to provide more accurate information. The function could accept a validator address as an input parameter. This would allow it to return the maximum amount a user can withdraw from a *specific* validator, aligning its logic with that of the `withdraw(...)` and `withdrawFromSpecificValidator(...)` functions and preventing misleading results.

Status: Acknowledged.

Update from the client: The `maxWithdraw(...)` function is designed to reflect the theoretical maximum a user can redeem based on their LST token balance, not validator-specific liquidity constraints. The per-validator limits are intentionally enforced at the withdrawal level and surfaced through the front end, which directs withdrawals to appropriate validators. As such, while the function may overstate the amount withdrawable from a specific validator, this is by design and does not impact protocol safety.

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 TruStake documentation

TruFin provides well-structured and detailed information on the functionality of the TruStake vaults. TruFin's team was also proactive in making themselves available for sync calls to address any questions or concerns raised during the review. These discussions ensured that the security team had all the necessary context to perform a thorough analysis.

8 Test Suite Evaluation

8.1 Tests Output

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

Ran 2 tests for test/Foundry/DelegateRegistry.t.sol:DelegateRegistryState
[PASS] testSetGovernanceDelegation() (gas: 33325)
[PASS] testSetGovernanceDelegationClearsDelegationWhenNoneSet() (gas: 36061)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 33.62ms (3.18ms CPU time)

Ran 10 tests for test/Foundry/Pause.t.sol:Pause
[PASS] testCannotClaimWithdrawFromSpecificValidatorWhenContractPaused() (gas: 55564)
[PASS] testCannotClaimWithdrawWhenContractPaused() (gas: 53382)
[PASS] testCannotClaimWithdrawalWhenContractPaused() (gas: 55578)
[PASS] testCannotCompoundingRewardsWhenContractPaused() (gas: 47880)
[PASS] testCannotDepositToSpecificValidatorWhenContractPaused() (gas: 55519)
[PASS] testCannotDepositWhenContractPaused() (gas: 53272)
[PASS] testCannotWithdrawClaimListWhenContractPaused() (gas: 56193)
[PASS] testContractCanOnlyBePausedByOwner() (gas: 18683)
[PASS] testContractCanOnlyBeUnpausedByOwner() (gas: 43724)
[PASS] testNormalFunctionalityResumesAfterUnpause() (gas: 174579)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 34.96ms (4.11ms CPU time)

Ran 9 tests for test/Foundry/ERC20.t.sol:ERC20
[PASS] testBalanceOfPostRewardAccrual() (gas: 49544)
[PASS] testTotalSupplyAlteredByRewardsAfterDeposit() (gas: 129161)
[PASS] testTotalSupplyNotAlteredByRewards() (gas: 20333)
[PASS] testTransferFrom() (gas: 64955)
[PASS] testTransferFromPostDeposit() (gas: 241370)
[PASS] testTransferMoreThanBalanceReverts() (gas: 23885)
[PASS] testTransferPostDeposit() (gas: 59033)
[PASS] testWithdrawRequestPostAccrualDecreasesTotalSupply() (gas: 194697)
[PASS] testWithdrawRequestPreAccrualDecreasesTotalSupply() (gas: 165368)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 35.16ms (4.19ms CPU time)

Ran 3 tests for test/Foundry/Calculations.t.sol:CalculationsTests
[PASS] testConvertToAssets() (gas: 173045)
[PASS] testConvertToShares() (gas: 168219)
[PASS] testSharePriceCalculation() (gas: 175903)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 35.19ms (5.59ms CPU time)

Ran 24 tests for test/Foundry/Setters.t.sol:SettersTest
[PASS] testSetDelegateRegistry() (gas: 45785)
[PASS] testSetDelegateRegistryToSameAddress() (gas: 45324)
[PASS] testSetDelegateRegistryWithNonOwnerReverts() (gas: 18165)
[PASS] testSetDelegateRegistryWithZeroAddressReverts() (gas: 16075)
[PASS] testSetEpsilon() (gas: 45665)
[PASS] testSetEpsilonToSameValue() (gas: 43093)
[PASS] testSetEpsilonWithNonOwnerReverts() (gas: 16420)
[PASS] testSetEpsilonWithTooHighEpsilonReverts() (gas: 15926)
[PASS] testSetFee() (gas: 44307)
[PASS] testSetFeeToSameValue() (gas: 43163)
[PASS] testSetFeeWithNonOwnerReverts() (gas: 16534)
[PASS] testSetFeeWithTooHighFeeReverts() (gas: 16083)
[PASS] testSetMinDeposit() (gas: 42845)
[PASS] testSetMinDepositToSameValue() (gas: 40361)
[PASS] testSetMinDepositWithNonOwnerReverts() (gas: 16411)
[PASS] testSetMinDepositWithTooLowMinDepositReverts() (gas: 15823)
[PASS] testSetTreasury() (gas: 45792)
[PASS] testSetTreasuryToSameAddress() (gas: 45187)
[PASS] testSetTreasuryWithNonOwnerReverts() (gas: 18093)
[PASS] testSetTreasuryWithZeroAddressReverts() (gas: 15940)
[PASS] testSetWhitelist() (gas: 45895)
[PASS] testSetWhitelistToSameAddress() (gas: 45370)
[PASS] testSetWhitelistWithNonOwnerReverts() (gas: 18216)
[PASS] testSetWhitelistWithZeroAddressReverts() (gas: 16051)
Suite result: ok. 24 passed; 0 failed; 0 skipped; finished in 35.26ms (5.57ms CPU time)

Ran 5 tests for test/Foundry/Restaking.t.sol:RestakeState
```

```
[PASS] testCompoundRewardsEmitsEvent() (gas: 124576)
[PASS] testCompoundRewardsEmitsEventForError() (gas: 61919)
[PASS] testCompoundRewardsWithDisabledStakerReverts() (gas: 67726)
[PASS] testTreasuryMintedRewardsOnDepositIfRestakingFails() (gas: 274896)
[PASS] testTreasuryOnlyMintedSharesForSuccessfulRestakes() (gas: 271703)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 35.50ms (4.42ms CPU time)

Ran 12 tests for test/Foundry/Deposit.t.sol:DepositTests
[PASS] testCanDepositMinDepositAmount() (gas: 159695)
[PASS] testDepositEmitsEvent() (gas: 154227)
[PASS] testDepositUpdatesUserInfo() (gas: 181464)
[PASS] testDepositUpdatesValidatorStruct() (gas: 163530)
[PASS] testDepositWithTooLittlePOLFails() (gas: 137076)
[PASS] testDepositsByMultipleUsers() (gas: 263775)
[PASS] testInflationAttackDoesNotWork() (gas: 216544)
[PASS] testMultipleDeposits() (gas: 202937)
[PASS] testRevertsWhenDepositingZeroPOL() (gas: 30756)
[PASS] testRevertsWithLessThanMinDepositAmount() (gas: 35346)
[PASS] testRevertsWithNonWhitelistedUser() (gas: 21281)
[PASS] testSingleDeposit() (gas: 169474)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 35.67ms (4.78ms CPU time)

Ran 6 tests for test/Foundry/Initialize.t.sol:InitializeTest
[PASS] testEmitsEvent() (gas: 397705)
[PASS] testInitializeAgainReverts() (gas: 397124)
[PASS] testInitializeSetsVariables() (gas: 403569)
[PASS] testRevertFeeTooLarge() (gas: 144764)
[PASS] testRevertWithZeroAddress() (gas: 770657)
[PASS] testStakerInitialValues() (gas: 410949)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 36.64ms (6.42ms CPU time)

Ran 1 test for test/Foundry/Validators.t.sol:GetAllValidatorsTests
[PASS] testReturnsAllValidators() (gas: 154130)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.74ms (223.58µs CPU time)

Ran 12 tests for test/Foundry/Deposit.t.sol:DepositToSpecificValidatorTests
[PASS] testCanDepositMinDepositAmount() (gas: 157800)
[PASS] testDepositMintsTreasurySharesForRewards() (gas: 218837)
[PASS] testDepositsByMultipleUsers() (gas: 262265)
[PASS] testDepositsByMultipleUsersToDifferentValidators() (gas: 287247)
[PASS] testIncreasesValidatorStake() (gas: 159587)
[PASS] testMultipleDeposits() (gas: 201355)
[PASS] testRevertsWhenDepositingZeroPOL() (gas: 33087)
[PASS] testRevertsWhenStakingToADisabledValidator() (gas: 44735)
[PASS] testRevertsWhenStakingToNonExistentValidator() (gas: 33355)
[PASS] testRevertsWithLessThanMinDepositAmount() (gas: 35675)
[PASS] testRevertsWithNonWhitelistedUser() (gas: 26209)
[PASS] testSingleDeposit() (gas: 167712)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 37.01ms (6.06ms CPU time)

Ran 1 test for test/Foundry/Validators.t.sol:GetValidatorsTests
[PASS] testIncludesDefaultValidator() (gas: 21299)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.26ms (29.75µs CPU time)

Ran 5 tests for test/Foundry/Getters.t.sol:MaxWithdrawTests
[PASS] testGetDustReturnsCorrectValue() (gas: 26722)
[PASS] testGetUnbondNonce() (gas: 40148)
[PASS] testInitialMaxWithdrawValue() (gas: 20999)
[PASS] testMaxWithdrawAfterDeposit() (gas: 158047)
[PASS] testPreviewFunctionCircularChecks() (gas: 927526)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 37.69ms (7.44ms CPU time)

Ran 6 tests for test/Foundry/Validators.t.sol:DisableValidatorTests
[PASS] testDisablesValidator() (gas: 32226)
[PASS] testEmitsEvent() (gas: 27474)
[PASS] testRevertsWhenCallerIsNotTheOwner() (gas: 18755)
[PASS] testRevertsWithDisabledValidatorAddress() (gas: 27956)
[PASS] testRevertsWithUnknownValidatorAddress() (gas: 18183)
[PASS] testRevertsWithZeroAddress() (gas: 15862)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 2.10ms (469.29µs CPU time)

Ran 2 tests for test/Foundry/TruPOL.t.sol:TruPOLTests
[PASS] testGetName() (gas: 17167)
[PASS] testGetSymbol() (gas: 17231)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.50ms (1.27ms CPU time)
```

```
Ran 5 tests for test/Foundry/Validators.t.sol:AddValidatorTests
[PASS] testAddsValidator() (gas: 82106)
[PASS] testEmitsEvent() (gas: 67738)
[PASS] testRevertsWhenCallerIsNotTheOwner() (gas: 16576)
[PASS] testRevertsWithExistingAddress() (gas: 20070)
[PASS] testRevertsWithZeroAddress() (gas: 15732)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 1.47ms (273.34µs CPU time)

Ran 6 tests for test/Foundry/Validators.t.sol:EnableValidatorTests
[PASS] testEmitsEvent() (gas: 27505)
[PASS] testEnablesValidator() (gas: 32302)
[PASS] testRevertsWhenCallerIsNotTheOwner() (gas: 18753)
[PASS] testRevertsWithEnabledValidatorAddress() (gas: 27874)
[PASS] testRevertsWithUnknownValidatorAddress() (gas: 18159)
[PASS] testRevertsWithZeroAddress() (gas: 15882)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 1.53ms (282.54µs CPU time)

Ran 5 tests for test/Foundry/Validators.t.sol:SetDefaultValidatorTests
[PASS] testEmitsEvent() (gas: 78775)
[PASS] testRevertsWhenCallerIsNotTheOwner() (gas: 18645)
[PASS] testRevertsWithNonEnabledValidator() (gas: 73214)
[PASS] testRevertsWithZeroAddress() (gas: 15990)
[PASS] testSetsTheDefaultValidator() (gas: 93235)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 2.08ms (214.37µs CPU time)

Ran 10 tests for test/Foundry/WithdrawClaim.t.sol:WithdrawClaimState
[PASS] testClaimListWhenNotWhitelistedReverts() (gas: 40542)
[PASS] testClaimListWithInvalidNoncesReverts() (gas: 35732)
[PASS] testClaimNonExistentWithdrawalReverts() (gas: 35220)
[PASS] testClaimWithdrawalFromDifferentUserReverts() (gas: 67284)
[PASS] testClaimWithdrawalTwiceReverts() (gas: 47002)
[PASS] testClaimWithdrawalWhenNotWhitelistedReverts() (gas: 40123)
[PASS] testIsClaimableReturnsFalseForNonClaimableWithdrawal() (gas: 27886)
[PASS] testIsClaimableReturnsFalseForNonExistentWithdrawal() (gas: 27861)
[PASS] testIsClaimableReturnsTrueForClaimableWithdrawals() (gas: 27741)
[PASS] testNoPOLReceivedByValidator() (gas: 47545)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 2.97ms (591.54µs CPU time)

Ran 12 tests for test/Foundry/Withdraw.t.sol:WithdrawTests
[PASS] testCanImmediatelyWithdraw() (gas: 200963)
[PASS] testCanWithdrawMaxAmountWhenSharePriceIsGreaterThanOne() (gas: 167159)
[PASS] testCanWithdrawMaxAmountWhenSharePriceIsOne() (gas: 165121)
[PASS] testCannotWithdrawMoreThanMaxAmountWhenSharePriceIsGreaterThanOne() (gas: 104044)
[PASS] testCannotWithdrawMoreThanMaxAmountWhenSharePriceIsOne() (gas: 99075)
[PASS] testFullWithdrawal() (gas: 168210)
[PASS] testMultiplePartialWithdrawals() (gas: 248798)
[PASS] testPartialWithdraw() (gas: 179578)
[PASS] testWithdrawMoreThanDepositedReverts() (gas: 62748)
[PASS] testWithdrawWhenNotWhitelistedReverts() (gas: 21268)
[PASS] testWithdrawZeroReverts() (gas: 30842)
[PASS] testWithdrawalWithRewards() (gas: 233228)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 3.45ms (2.70ms CPU time)

Ran 6 tests for test/Foundry/Withdraw.t.sol:WithdrawFromSpecificValidator
[PASS] testFullWithdrawal() (gas: 180551)
[PASS] testPartialWithdraw() (gas: 190703)
[PASS] testTreasuryOnlyMintedSharesForClaimedRewards() (gas: 242543)
[PASS] testWithdrawFromNonExistentValidatorReverts() (gas: 31172)
[PASS] testWithdrawWhenNotWhitelistedReverts() (gas: 23631)
[PASS] testWithdrawalWithAmountAboveValidatorStakeReverts() (gas: 587784)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 4.71ms (2.68ms CPU time)

Ran 4 tests for test/Foundry/utils/StorageUtilsTest.t.sol:StorageUtilsTest
[PASS] testFee() (gas: 5987)
[PASS] testFuzzBalanceOf(address,uint256) (runs: 256, : 255111, ~: 255108)
[PASS] testTotalSupply() (gas: 132824)
[PASS] testTreasuryAddress() (gas: 7464)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 126.86ms (97.05ms CPU time)

Ran 1 test for test/Foundry/invariant/SharePriceInvariant.t.sol:SharePriceInvariantTest
[PASS] invariant_SharesPriceDoesNotChange() (runs: 10, calls: 1000, reverts: 0)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 141.88s (134.81s CPU time)

Ran 1 test for test/Foundry/invariant/TotalCapitalInvariant.t.sol:TotalCapitalInvariantTest
```

```
[PASS] invariant_SharesValueMatchesCapital() (runs: 10, calls: 1000, reverts: 0)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 161.35s (154.31s CPU time)

Ran 23 test suites in 161.41s (303.73s CPU time): 148 tests passed, 0 failed, 0 skipped (148 total tests)
```

8.2 Automated Tools

8.2.1 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 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