Security Review Report
NM-0098 TruFin

NETHERMIND

(Jun 13, 2023)

Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind for the TruFin protocol. TruFin provides access to MATIC staking on Ethereum for institutions featuring auto-compounding of rewards and the ability to allocate rewards to other addresses. The protocol is permissioned, ensuring that each user has passed know-your-customer (KYC) checks and that only whitelisted users can stake. A user's stake is represented by ERC4626 tokens which track the increase in value from the accrual of rewards.

The audited code comprises 1,324 lines of Solidity, with a code test coverage of 97.22% for the primary contract `TruStakeMATICv2`. The `TruFin` team has provided detailed documentation explaining the protocol summary, deposit/withdraw flows, precision and rounding choices, and rewards allocation/distribution.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contracts, and (d) creation of test cases. Along this document, we report 11 points of attention, where 2 are classified as `Medium`, 1 is classified as `Low`, and 8 are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 discusses the risk rating methodology adopted for this audit. Section 5 details the issues. Section 6 discusses the documentation provided by the client for this audit. Section 7 presents the compilation, tests, and automated tests. Section 8 concludes the document.

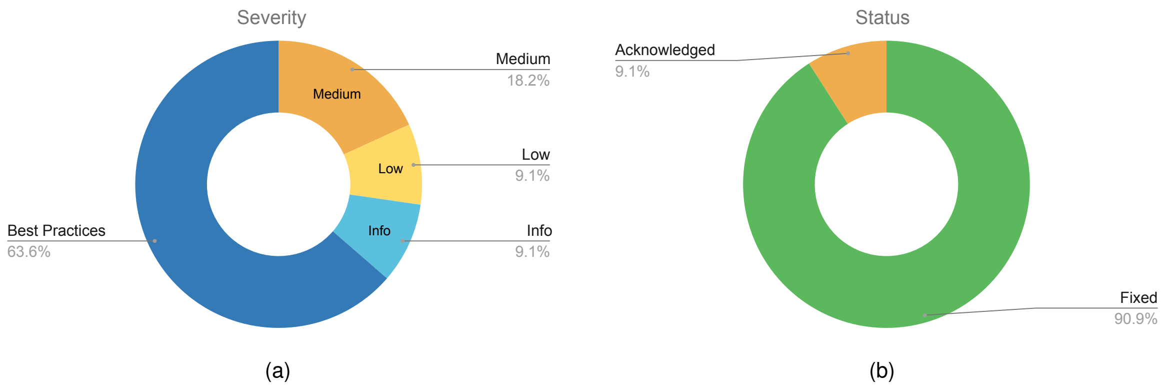

Fig 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (1), **Undetermined** (0), **Informational** (1), **Best Practices** (7).
Distribution of status: Fixed (10), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	June 5, 2023
Response from Client	June 12, 2023
Final Report	June 13, 2023
Methods	Manual Review, Automated Analysis
Repository	https://github.com/TruFin-io/staker-audit-april/
Commit Hash (Initial Audit)	e5065cb2e3c7c49e28d0644b1128eae0fa4d5a75
Branch (Review Audit)	Branch: fixes-1
Documentation	Provided protocol documentation (PDFs)
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	main/TruStakeMATICv2.sol	915	412	45.0%	279	1606
2	main/TruStakeMATICv2Storage.sol	28	28	100.0%	20	76
3	helpers/MasterWhitelist.sol	295	258	87.5%	85	638
4	interfaces/IValidatorShare.sol	15	3	20.0%	13	31
5	interfaces/IStakeManager.sol	55	2	3.6%	21	78
6	interfaces/IMasterWhitelist.sol	16	39	243.8%	10	65
	Total	**1324**	**742**	**56.0%**	**428**	**2494**

3 Summary of Issues

	Finding	Severity	Update
1	Changing `stakingTokenAddress` will affect share pricing and staking calculations	Medium	Fixed
2	Inflation attack can lead to loss of value for depositors	Medium	Fixed
3	The `setCap(...)` function can set a cap lower than the current deposited amount	Low	Fixed
4	Shadowed variables and functions	Info	Fixed
5	Boolean comparisons can be avoided	Best Practices	Fixed
6	Usage of outdated solidity compiler	Best Practices	Fixed
7	Using enum instead of arbitrary number for `userType` .	Best Practices	Acknowledged
8	Incorrect comment in `reallocate(...)`	Best Practices	Fixed
9	NatSpec documentation refers to outdated variables	Best Practices	Fixed
10	Solidity code style	Best Practices	Fixed
11	Unnecessary storage write in `deallocate(...)`	Best Practices	Fixed

4 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 an attacker will uncover and exploit the finding. 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 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 formally pass to the client;

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.

5 Issues

5.1 [Medium] Changing stakingTokenAddress will affect share pricing and staking calculations

File(s): TruStakeMATICv2.sol

Description: In the TruStakeMATICv2 contract, it is possible to change the staking token through the function setStakingToken(...). However, it does not consider that the ERC4626 private storage variable _asset won't be changed. Most original functions that rely on _asset have been overridden to use stakingTokenAddress, except for totalAssets(...). This means that when a new staking token is used, totalAssets(...) will return the balance count for the original token used during initialization. The following functions use totalAssets(...) and may be affected:

- sharePrice(...);
- _deposit(...);
- _withdrawRequest(...);
- compoundRewards(...);

Incorrect reported staking token balances through totalAssets(...) may affect share pricing and rewards and staking calculations.

Recommendation(s): Consider overriding the totalAssets(...) function to return the balance of stakingTokenAddress instead of _asset.

Status: Fixed

Update from the client: Setting the token address is now only possible at initialization time to prevent the issues evoked above.

5.2 [Medium] Inflation attack can lead to loss of value for depositors

File(s): TruStakeMATICv2.sol

Description: In the TruStakeMATICv2 contract, the initial depositor can use an inflation attack to cause the second depositor to pay more than expected for their shares, leading to a loss of value. The steps to manipulate the pricing are shown below:

1. Attacker makes an initial deposit of 1 MATIC using the function deposit(. . .);
2. Attacker immediately requests for a withdrawal of 0.9 MATIC, leaving a small amount remaining;
3. Before the second depositor starts their deposit, the attacker directly transfers a large amount of MATIC to the contract (without depositing);

After these steps, the second deposit will calculate pricing based on a higher share price since part of the pricing calculation is to check the balance of assets on the contract, which has been manipulated due to a direct transfer.

Recommendation(s): Consider establishing a minimum remaining balance amount after withdrawal, and if a withdrawal leads to a remaining amount less than this minimum, then withdraw the entire user's balance.

Status: Fixed

Update from the client: We fixed this issue by not allowing users to leave small amounts in the vault.

5.3 [Low] The setCap(...) function can set a cap lower than the current deposited amount

File(s): TruStakeMATICv2.sol

Description: The function setCap(...) is missing input validation to prevent the cap from being lower than the current deposited amount. Without this check, the following logic in maxDeposit(...) could revert due to an underflow.

```
function maxDeposit(address) public view override returns (uint256) {
    return cap - totalStaked();
}
```

Recommendation(s): Consider adding input validation to setCap(...) to prevent a cap smaller than the deposit amount from being set.

Status: Fixed

Update from the client: We check the cap is always set to be higher than the totalStaked.

5.4 [Info] Shadowed variables and functions

File(s): TruStakeMATICv2.sol

Description: The TruStakeMATICv2 contract contains some functions which have variables with the same name as storage variables or functions from other imported contracts. This can lead to confusion where it may be unclear which data the variables are actually referring to. A list of all the shadowing variables are presented below:

- distributeAll(...): Variable totalAllocated shadows storage variable TruStakeMATICv2Storage.totalAllocated;
- maxWithdraw(...): Argument owner shadows function OwnableUpgradeable.owner(...);
- maxRedeem(...): Argument owner shadows function OwnableUpgradeable.owner(...);
- getUserInfo(...): Argument _owner shadows storage variable OwnableUpgradeable._owner;
- withdraw(...): Argument owner shadows function OwnableUpgradeable.owner(...);
- redeem(...): Argument owner shadows function OwnableUpgradeable.owner(...);

Recommendation(s): Consider changing the names of the variables and arguments listed above to ensure they do not have the same name as existing inherited functions or storage variables.

Status: Fixed

Update from the client: All shadowing variables have been renamed.

5.5 [Best Practice] Boolean comparisons can be avoided

File(s): TruStakeMATICv2.sol

Description: In the function allocate(...) boolean comparisons are used as below:

```
function allocate(
    uint256 _amount,
    address _recipient,
    bool _strict
) external onlyWhitelist nonReentrant {
    if (_strict == true && allowStrict == false) {
        revert StrictAllocationDisabled();
    }
    ...
}
```

Recommendation(s): Boolean comparisons can be simplified. The above can be simplified as below.

```
if (_strict && !allowStrict) {
    revert StrictAllocationDisabled();
}
```

Status: Fixed

Update from the client: We removed the boolean comparison per the recommendation.

5.6 [Best Practice] Usage of outdated solidity compiler

File(s): contracts

Description: Solidity pragma version 0.8.14 is used in all contracts. Version 0.8.14 has some known optimizer-related bugs.

Recommendation(s): Consider using an updated and stable version like 0.8.18 of solidity.

Status: Fixed

Update from the client: The compiler has been updated to 0.8.19.

5.7 [Best Practice] Using enum instead of arbitrary number for userType.

File(s): MasterWhitelist.sol

Description: In the contract MasterWhitelist, the userTypes in the events are declared by arbitrary numbers. It is mentioned in the comments:

```
* @param userType can take values 0,1,2,3,4 if the address is a user, market maker, vault, lawyer, or swapManager
↪  respectively
```

But it would be easier for readability to have enums instead of numbers to describe the user types and less prone to mistakes.

Recommendation(s): Consider using enums that describe the type of user.

Status: Acknowledged

Update from the client: Other parts of the system (on-chain and off-chain) are relying on these numbers today, so changing them will require a more coordinated effort on our part. We're also planning to add more KYC providers so we'll make these changes at that time.

5.8 [Best Practices] Incorrect comment in reallocate(...)

File(s): TruStakeMATICv2.sol

Description: In TruStakeMATICv2, the function reallocate(...), if you are reallocating to a user that you already have an allocation for, the recipients array, is searched for the _oldRecipient and replaced by the _newRecipient by directly overwriting. However, the comment appears to be copied from a different section of the code, as it refers to popping an item in the array, which does not happen in this part of the code.

Recommendation(s): Consider updating the comment to accurately reflect the approach to updating the recipients array.

Status: Fixed

Update from the client: This comment has been updated, along with other comments in the contract, to more accurately reflect the code.

5.9 [Best Practices] NatSpec documentation refers to outdated variables

File(s): TruStakeMATICv2.sol

Description: In TruStakeMATICv2 the function sharePrice(...) has NatSpec documentation that refers to two return variables priceNum and priceDenom. However, the variable names and all other returned data from sharePrice(...) refer to the variables as globalPriceNum and globalPriceDenom.

```
/// @return priceNum numerator of share price, divide by (priceDenom * 1e18) to get actual floating point share price
/// @return priceDenom denominator of share price
function sharePrice() public view returns (uint256, uint256) {
    ...
    ///////////////////////////////////////////////////////////////////////
    // @audit Returned variable names do not align with NatSpec
    ///////////////////////////////////////////////////////////////////////
    uint256 globalPriceNum = totalCapitalTimesPhiPrecision * 1e18;
    uint256 globalPriceDenom = totalSupply() * phiPrecision;
    return (globalPriceNum, globalPriceDenom);
}
```

Recommendation(s): Consider updating the NatSpec documentation for sharePrice(...) to accurately reflect the return variable names.

Status: Fixed

Update from the client: All NatSpec docs have been updated.

5.10 [Best Practices] Solidity code style

File(s): FileName

Description: The following is a collection of Solidity code style inconsistencies that do not align with the current recommended style guide.

– The constant phiPrecision should be all uppercase with underscores separating words;

Recommendation(s): Consider adjusting the code to follow the Solidity style guide.

Status: Fixed

Update from the client: We formatted the contract to be more consistent with the Solidity style guide.

5.11 [Best Practices] Unnecessary storage write in `deallocate(...)`

File(s): `TruStakeMATICv2.sol`

Description: In `TruStakeMATICv2`, the function `deallocate(...)` has a check to see if the callers `totalAllocated` amount needs to be cleared entirely or reduced and adjusted. After this `if/else` block, `totalAllocated` updated. When `totalAmount` is zero, the entry is deleted (set to zero) and then set to zero again outside of the `if/else` block, which is an unnecessary storage write. A code snippet is shown below.

```
uint256 totalAmount;
uint256 totalPriceNum;
uint256 totalPriceDenom;
Allocation storage totalAllocation = totalAllocated[msg.sender][_strict];
totalAmount = totalAllocation.maticAmount - _amount;

if (totalAmount == 0) {
    delete totalAllocated[msg.sender][_strict];
} else {
    // Calculate new `totalPriceNum` and `totalPriceDenom`
}

////////////////////////////////////////////////////////////////////
// @audit When `totalAmount == 0`, `totalAllocated` is cleared twice
////////////////////////////////////////////////////////////////////
totalAllocated[msg.sender][_strict] = Allocation(totalAmount, totalPriceNum, totalPriceDenom);
```

Recommendation(s): Consider moving the `totalAllocated` update within the `else` of the `if/else` block to avoid a double write when `totalAmount` is zero.

Status: Fixed

Update from the client: This has been fixed in line with the recommendation.

6 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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.

The `TruFin` team has provided extensive `documentation` on their GitHub repo, including an overview of the protocol and technical insights into its components. A summary of these documents is shown below:

- General Overview
- Variable and Mappings
- Precision
- Deposits and Withdrawals
- Allocation
- Distribution
- Front end values
- Off-chain (web2) sanity checks

Each function is complemented by detailed NatSpec comments describing function behavior, inputs, and outputs. Moreover, the team conducted a comprehensive code walkthrough and maintained open communication to address any inquiries or concerns raised by the Nethermind auditors.

7 Test Suite Evaluation

7.1 Contracts Compilation Output

```
> npx hardhat compile

Downloading compiler 0.8.14
Generating typings for: 22 artifacts in dir: typechain-types for target: ethers-v5
Successfully generated 76 typings!
Compiled 19 Solidity files successfully
```

7.2 Tests Output

```
  INIT
    INITIALISATION
       global variables initialised with correct values (64ms)
       validating initializer parameters (66ms)
    MODIFIERS
       onlyWhitelist
    SETTERS - events and ownable
       setValidatorShareContract (47ms)
       setWhitelist (63ms)
       setTreasury (46ms)
       setCap (55ms)
       setEpsilon (40ms)
       setPhi (46ms)
       setDistPhi (41ms)
       owner successfully sets allowStrict flag
       non-owner setting allowStrict flag fails
       owner successfully sets epsilon
       non-owner setting epsilon fails
    ATTACKS
       inflation frontrunning attack investigation (274ms)
       fail: depositing under 1 matic
       pass: successfully deposit 1 matic or more (401ms)

  GETTERS
    ERC-4626: max functions
       maxDeposit (273ms)
       maxMint (299ms)
       maxWithdraw (239ms)
       pass: output of maxWithdraw is greater than to just deposited amount without accrual (245ms)
       pass: withdraw output of maxWithdraw after depositing (607ms)
       fail: cannot withdraw 1 + output of maxWithdraw after depositing (257ms)
       pass: withdraw output of maxWithdraw after depositing and accruing rewards (5545ms)
       fail: cannot withdraw 1 + output of maxWithdraw after depositing and accruing rewards (4882ms)
       maxRedeem (164ms)
       preview functions circular check (36558ms)
    ERC-4626: getters + metadata
       name
       symbol

  SETTERS
    setValidatorShareContract
       Reverts with zero address
       Works with a new address
       Works with the same address
    setWhitelist
       Reverts with zero address
       Works with a new address
       Works with the same address
    setTreasury
       Reverts with zero address
       Works with a new address
       Works with the same address
    setPhi
       General input validation; reverts when too high
       Works with a new value
       Works with the same value
    setDistPhi
       General input validation; reverts when too high
       Works with a new value
       Works with the same value
    setCap
       Reverts with too low value (186ms)
       Works with a new value
       Works with the same value
    setEpsilon
       Reverts with too high value
       Works with a new value
       Works with the same value

  Other
    allocate
       Reverts with zero address (172ms)

  Deployment
    INITIALISATION
       Reverts on zero address (200ms)

  Checkpoint Submissions
     use submitCheckpoint multiple times, ensure rewards are increasing each time (23509ms)

  DEPOSIT
     single deposit (221ms)
     repeated deposits (398ms)
     multiple account deposits (603ms)
     deposit zero matic (202ms)
     try depositing more than the cap
     Can withdraw maxWithdraw amount (26240ms)
     can immediately withdraw deposited amount (463ms)
     unknown non-whitelist user deposit fails
     unknown non-whitelist user cannot deposit to a whitelisted user's address
     user cannot drain vault by depositing zero (127ms)

  WITHDRAW REQUEST
     initiate a partial withdrawal (382ms)
     initiate a complete withdrawal (380ms)
     initiate multiple partial withdrawals (536ms)
     initiate withdrawal with rewards wip (5123ms)
     try initiating a withdrawal of size zero
     try initiating withdrawal of more than deposited (221ms)
     Withdraw of strictly-allocated funds blocked (299ms)

  WITHDRAW CLAIM
    User: withdrawClaim
       try claiming withdrawal requested by different user (42ms)
       try claiming withdrawal requested 79 epochs ago (53ms)
       try claiming withdrawal with unbond nonce that doesn't exist
       try claiming already claimed withdrawal (69ms)
       successfully claim withdrawal requested 80 epochs ago with expected changes in state and balances (111ms)
    User: claimList
       try to claim test unbonds when one has not matured (101ms)
       try to claim test unbonds when one has already been claimed (71ms)
       try to claim test unbonds when one has a different user (87ms)
       successfully claim three test unbonds consecutively (145ms)
       successfully claim two of three test unbonds inconsecutively (105ms)
       successfully claim just one withdrawal (75ms)

  RESTAKE
    Vault: Simulate rewards accrual
       Simulating `SubmitCheckpoint` transaction on RootChainProxy (23226ms)
    Vault: compound rewards
       rewards compounded correctly (compoundRewards: using unclaimed rewards) (14233ms)
       rewards compounded correctly (stakeClaimedRewards: using claimed rewards) (6318ms)
       try compounding rewards with rewards equal to zero (216ms)
```

```
ERC-4626 (TEMP)
  ERC-4626: standard share exchange functions
     deposit (270ms)
     mint (296ms)
     withdraw (401ms)
     pass: redeem some shares to oneself works (479ms)
     pass: redeem entire staked balanceOf to oneself (545ms)

ALLOCATION
  LOOSE
     pass: making two allocations adding up to MATIC amount larger than user deposited MATIC (138ms)
     pass: allocating twice adds up to allocated amount correctly (134ms)
     pass: first allocation updates state accordingly (127ms)
     pass: multiple allocations update share prices in allocation mappings correctly (23823ms)
     pass: multiple allocations to different people work correctly (5233ms)
     multiple allocations to different people at different shareprices work correctly (18626ms)
     pass: should be able to transfer loosely allocated balance (98ms)
     pass: allocate full amount deposited to one other user (250ms)
     pass: overallocation: allocate 3x more than deposited, recipient rewards math is unchanged, distributors deposit
     ↪  amount is reduced to cover all rewards (25755ms)
     pass: distributing rewards does not affect allocated amount (23555ms)
     pass: repeated allocation to the same recipient (with accrual of rewards in between), computes new combined
     ↪  allocation share price correctly (23753ms)
     dump, deposit, allocate, accrue, distribute, withdraw deposited returns correct getUserInfo (5423ms)
     deposit, allocate, accrue, distribute, withdraw deposited returns correct getUserInfo (19200ms)
     fail: distributing rewards fails if distributor has transferred deposited amount beforehand (4902ms)
     fail: allocating more than deposited at once (46ms)
     fail: allocating zero (40ms)
  STRICT
     pass: allocating twice adds up to allocated amount correctly (141ms)
     pass: first allocation updates state accordingly (123ms)
     pass: multiple allocations update share prices in allocation mappings correctly (23642ms)
     pass: multiple allocations to different people work correctly (5339ms)
     multiple allocations to different people at different shareprices work correctly (18669ms)
     pass: repeated allocation to the same and other recipients (with accrual of rewards in between && artificially
     ↪  doubled share price), calculates rewards correctly (23171ms)
     pass: distributing rewards works correctly for allocations (23318ms)
     pass: allocate full amount deposited to one other user (240ms)
     pass: allocate full amount deposited to one other user, rounding error (23669ms)
     increase spx using dumping (227ms)
     dump, deposit, allocate, accrue, distribute, withdraw deposited amt (23530ms)
     deposit, allocate, accrue, distribute, withdraw deposited amt (23388ms)
     fail: try allocating strictly when `allowStrict` is set to false
     fail: allocating more than deposited (138ms)
     fail: allocating zero (39ms)
     fail: should be unable to transfer strictly allocated balance (80ms)
     pass: checks that you can strictly allocate entire maxWithdraw amount (including epsilon) (97ms)
     maxRedeemable in getUserInfo returns maxWithdraw when the max balance is strictly allocated (163ms)
  BOTH
     pass: mix of strict & loose allocation update state accordingly (267ms)
     pass: mix of loose & strict allocation update state accordingly (opposite of previous case) (276ms)

DEALLOCATE
  LOOSE
     Emits 'Deallocated' event with expected parameters (60ms)
     Reverts if caller has not made an allocation to the input recipient (38ms)
     Reverts via underflow if deallocated amount larger than allocated amount
     Removes recipient from distributor's recipients if full individual deallocation (324ms)
     Removes distributor from recipient's distributors if full individual deallocation (326ms)
    Individual Allocation State
       Individual allocation price is not changed during deallocation (52ms)
       Reduces individual allocation by deallocated amount (49ms)
       Deletes individual allocation from storage if full individual deallocation (50ms)
    Total Allocation State
       Deletes total allocation from storage if complete total deallocation (49ms)
       Updates total allocation price if partial total deallocation (50ms)
       Decreases total allocation amount if partial total deallocation (51ms)
    Functionality
       Deallocate reduces rewards proportionally (5014ms)
       Deallocation leads to rewards if the reduced amount was allocated initially (before any distribution) (4927ms)
       Non-strict flow of allocating, deallocating and distributing as rewards accrue (18935ms)
```

```
   STRICT
      Simple deallocation pre reward accrual (51ms)
      Simple deallocation updates mappings correctly (62ms)
      Pending rewards are distributed upon deallocation (4835ms)
      Partial deallocation: share price updated correctly (4718ms)
      Partial deallocation: totalAllocated share price updated correctly (4905ms)
      Full deallocation: mappings updated correctly (5107ms)
      Strict flow of allocating, deallocating and distributing as rewards accrue (20438ms)

 REALLOCATE
    pass: reallocation to empty allocation (5559ms)
    pass: reallocation to existing allocation (from older allocation to more recent allocation) (5519ms)
    pass: reallocation to existing allocation (from more recent allocation to older allocation) (5525ms)
    pass: reallocating to yourself should be possible (5307ms)
    pass: reallocating to a non-whitelisted user should be possible (5401ms)
    fail: reallocate from non-existent allocation
    reallocating strict allocation fails (242ms)

 DISTRIBUTION
   External Methods
     distributeRewards
        Reverts if target allocation is loose and caller is not the allocator
        Allows calls from non-allocator addresses if allocation is strict (5091ms)
     distributeAll
        Reverts if target allocations are loose and caller is not the allocator
        Allows calls from non-allocator addresses if allocation is strict (5222ms)
        Calls _distributeRewards for all allocator's recipients (191ms)
        Updates distributor's total allocation price to current global price (176ms)
   Internal Methods
     _distributeRewards
        Emits 'DistributedRewards' with correct parameters inside distributeAll call (124ms)
        Transfers rewards as TruMATIC to recipient (114ms)
        Transfers TruMATIC recipientOneTruMATICFee to treasury (111ms)
        Updates individual price allocation (101ms)
        Strict rewards can be distributed with insufficient max redemption (5484ms)
     _distributeRewardsUpdateTotal
        Reverts if no allocation made by distributor to input recipient
        Skips reward distribution if global share price same as individual share price (52ms)
        Updates price of distributor's total allocation (5042ms)
        Emits 'DistributedRewards' event with correct parameters (4977ms)
   LOOSE
      Rewards earned via allocation equal rewards earned via deposit (5360ms)
      Can withdraw allocated amount after distributeRewards call (5584ms)
      Can withdraw combined allocated amounts after distributeAll call (10317ms)
      Mutliple distributeRewards calls are equivalent to single distributeAll call (11372ms)
   STRICT
      Rewards earned via strict allocation equal rewards earned via deposit (5460ms)
      Cannot withdraw strictly allocated amount after distributeRewards call (5189ms)
      Cannot withdraw combined allocated amounts after strict allocation (5462ms)
      Cannot withdraw combined allocated amounts after distributeAll call (5308ms)
      Mutliple distributeRewards calls are equivalent to single distributeAll call (11071ms)

 STRICT ALLOCATIONS
    Strict allocation attempts revert if allowStrict is false
    Allocation limit reduced by current total strict allocation (108ms)
    Updates individual strict allocation amount and price (88ms)
    Updates total strict allocation amount and price (87ms)
    Emits Allocation event with strict = true (78ms)
    Allocator added to recipient's strict distributors upon strict allocation (75ms)
    Recipient added to strict allocator's recipients upon strict allocation (77ms)

 MULTI CHECKPOINTS
    Lifecycle testing: Depositor amount+shares are locked as rewards accrue (strict allocations) and unlocked after
    ↪  deallocate, Receiver correctly aggregates shares+rewards across multiple checkpoints and after deallocate call,
    ↪  Depositor and receiver can withdraw max withdraw amount, treasury cannot due to epsilon (26219ms)
    Rewards are distributed correctly after _reallocation_ and _distribution_, reallocation after 1/2 the time is the
    ↪  same as having the allocation from the start: 100% to receiver, 0% rewards to depositor (10715ms)
    Reallocate, deallocate a strict and a loose allocation (without calling distribute), forces distribution of rewards
    ↪  in case of strict. for loose it is the same as having had no allocation (10637ms)
    Invariant testing: allocating strictly and loosely across two sets of users. Same workflow accross two separate
    ↪  user groups accrues the same amount of rewards  (11530ms)
```

```
TruMATIC ERC20 Functionality
   has a name
   has a symbol
  totalSupply
     totalSupply equals totalStaked for first deposits
     totalSupply is not altered by rewards accrual without deposit (5128ms)
     new deposit after reward accrual increases totalSupply (5322ms)
     withdraw requests pre accrual decrease totalSupply correctly (382ms)
     withdraw requests post accrual decrease totalSupply correctly (5724ms)
  balanceOf
     correctly updates balances post deposit (220ms)
     correctly updates sharePrice post reward accrual (5182ms)
  transfer
     correctly transfers post deposit (237ms)
     Reverts with custom error if more than users balance is transferred
     Transfer post loose allocation works (282ms)
  transferFrom
     Reverts without allowance/with insufficient balance (40ms)
     transferFrom after deposit works (249ms)
     transferFrom after loose allocation works (291ms)
  Strict Allocation
    totalSupply
       distributing rewards does not affect totalSupply (5345ms)
    transfer
       Transfer post strict allocation fails (306ms)
    transferFrom
       transferFrom after strict allocation reverts if more than unallocated balance is transferred (312ms)

ERC-4626
  deposit
     should revert if receiver is not caller
     should mint fresh shares to depositor (169ms)
    - should increase vault assets by deposited MATIC
     should emit 'Deposit' event (157ms)
  mint
     should revert if receiver is not caller
     should mint fresh shares to depositor (146ms)
     should emit 'Deposit' event (136ms)
  withdraw
     should burn shares from withdrawer (192ms)
     should emit 'Withdraw' event (178ms)
  redeem
     should revert if receiver is not caller
     should burn redeemed shares from redeemer's balance (197ms)
  maxRedeem
     should reduce maximum share redepmtion by strictly allocated amount (251ms)
     should not reduce maximum share redemption by loosely allocated amount (216ms)

Staker
  Inflation attack check
     Basic inflation attack (352ms)

Inflation Attack
  Checks
     Check Inflation not possible with user leaving tiny remaining balance (377ms)

Staker
  Scenario 1: transfer of allocated shares
     Scenario (628ms)

Scenario -- Check storage after allocate/deallocate/reallocate
  Flow
     Deposit as user1, deployer (348ms)
     Allocate user1 -> user2 (133ms)
     Allocate user1 -> deployer (140ms)
     Deallocate half user1 -> user2 (97ms)
     Deallocate last half user1 -> user2 (73ms)
     Reallocate user1 (deployer -> user2) (80ms)
     Allocate user1 -> deployer again (128ms)
     Reallocate user1 (user2 -> deployer) (63ms)
Staker
  Setters
     Set validator share contract
     Set whitelist
     Set treasury
     Set cap
     Set phi
     Set dist phi
  Main
     Deposit (194ms)
     Mint (200ms)
     Withdraw (379ms)
     Redeem (378ms)
  Allocations
     Allocate to user (500ms)
     Deallocate from user (322ms)
     Reallocate to another user (292ms)
     Distribute rewards (642ms)
  Revert
     When try to initialize contract again
     When not owner tries to set validator share contract
     When not owner tries to set whitelist
     When not owner tries to set treasury
     When not owner tries to set cap
     When not owner tries to set phi
     When not owner tries to set dist phi
     When try to set phi if it is too large
     When try to set dist phi if it is too large
     When not whitelisted user tries to allocate
     When not whitelisted user tries to deallocate
     When not whitelisted user tries to reallocate
     When try to allocate 0 amount (39ms)
     When try to deallocate
     When try to reallocate if allocation not exists
     When not whitelisted user tries to deposit
     When not whitelisted user tries to mint
     When not whitelisted user tries to withdraw
     When not whitelisted user tries to redeem


263 passing (15m)
1 pending
```

7.3 Code Coverage

```
> npx hardhat coverage
```

The relevant output is presented below.

```
-----------------------------|----------|----------|----------|----------|----------------|
File                         |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-----------------------------|----------|----------|----------|----------|----------------|
 helpers/                    |    14.55 |     8.33 |    13.64 |    17.82 |                |
  MasterWhitelist.sol        |    14.55 |     8.33 |    13.64 |    17.82 |... 628,629,631 |
 interfaces/                 |      100 |      100 |      100 |      100 |                |
  IMasterWhitelist.sol       |      100 |      100 |      100 |      100 |                |
  IStakeManager.sol          |      100 |      100 |      100 |      100 |                |
  ITruStakeMATICv2.sol       |      100 |      100 |      100 |      100 |                |
  IValidatorShare.sol        |      100 |      100 |      100 |      100 |                |
 main/                       |      100 |    86.59 |      100 |    97.22 |                |
  TruStakeMATICv2.sol        |      100 |    86.59 |      100 |    97.22 |... 707,773,774 |
  TruStakeMATICv2Storage.sol |      100 |      100 |      100 |      100 |                |
  Types.sol                  |      100 |      100 |      100 |      100 |                |
-----------------------------|----------|----------|----------|----------|----------------|
All files                    |    84.12 |    62.71 |    60.42 |    79.83 |                |
-----------------------------|----------|----------|----------|----------|----------------|
```

7.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

8 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.

Disclaimer