Security Review Report
NM-0268 Worldcoin Vault

NETHERMIND SECURITY

(July 23, 2024)

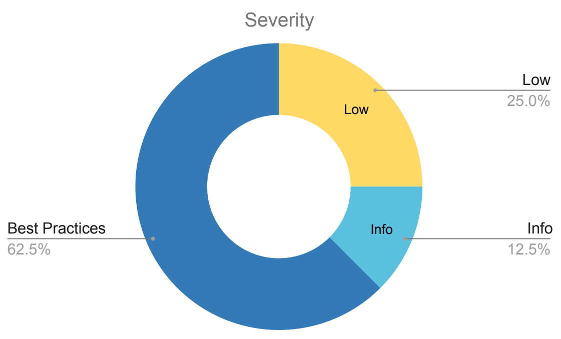
Contents

1 Executive Summary

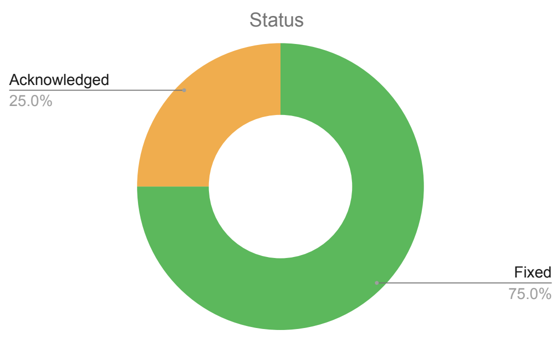
This document outlines the security review conducted by Nethermind Security for the Worldcoin Vault. The Vault is a smart contract allowing World App users to deposit `WLD` tokens and earn interest for their deposited amounts. Interest accrual is calculated based on the time elapsed since the deposit or the last claimed rewards. Furthermore, the contract integrates with the `WorldIDAddressBook`, which maintains a registry of verified users alongside with their verification time. Rewards are only accrued during periods when users are verified.

The audited code comprises 392 lines of code in Solidity. The **Worldcoin** team has provided inline documentation that explains the purpose and functionality behind the audited contracts. The audit was performed using: (a) manual analysis of the codebase, (b) simulation of the smart contracts. **Along this document, we report** 8 points of attention, where two are classified as `Low`, and six 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 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. 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.



(a) distribution of issues according to the severity



(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (1), **Best Practices** (5). **(b) Distribution of status: Fixed** (6), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	July 19, 2024
Final Report	July 23, 2024
Methods	Manual Review
Repository	worldcoin-vault
Commit Hash	7e24810f0bf467dc68b2aead2d22de981af33337
Final Commit Hash	28171872a23a03bef49898eb939934765157fa4b
Documentation	Inline comments
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	WLDVault.sol	226	111	49.1%	86	423
2	WorldIDAddressBook.sol	107	75	70.1%	39	221
3	interfaces/IVault.sol	40	34	85.0%	13	87
4	interfaces/IAddressBook.sol	13	11	84.6%	2	26
5	utils/ByteHasher.sol	6	5	83.3%	1	12
	Total	**392**	**236**	**60.2%**	**141**	**769**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect check in `_withdraw(...)` function could prevent valid withdrawals	Low	Fixed
2	Potential interest loss due to Address Book and verification length changes	Low	Fixed
3	Payable `verify(...)` function may lead to locked Ether	Info	Acknowledged
4	Avoid the usage of magic numbers	Best Practices	Fixed
5	Missing inputs validation	Best Practices	Acknowledged
6	Redundant check in `refresh(...)` function	Best Practices	Fixed
7	The `WalletVerified` event is not emitted in the `verify(...)` function	Best Practices	Fixed
8	The `verify(...)` function does not follow C-E-I pattern	Best Practices	Fixed

4 System Overview

The WLDVault enables World App users to deposit WLD tokens and earn yield over time. It interacts with the WorldIDAddressBook contract, which maintains a registry of verified users along with their verification timestamps. This ensures that only verified users can accrue interest based on their deposits.

4.1 WorldIDAddressBook

The WorldIDAddressBook contract is responsible for the verification of user addresses that can earn yield in WLDVault. This contract integrates with the WorldID proof verification system to authenticate users using zero-knowledge proofs.

The WorldIDAddressBook contract is initialized with several key parameters that control the verification process. These parameters are set during the contract's deployment:

- worldIdRouter: The instance of the WorldID router responsible for managing groups and verifying zero-knowledge proofs.
- groupId: A unique identifier for the group within the WorldID system.
- externalNullifierHash: A hash used to prevent double-signaling by ensuring each proof is unique.
- verificationLength: The duration, in seconds, for which a verification is considered valid.
- maxProofTime: The maximum allowable time, in seconds, in which a proof must be validated to be considered legitimate.

Users submit their verification proof through the verify(...) function, providing necessary details such as account address, nullifier hash, proof, and proof time.

```
function verify(
    address account,
    uint256 root,
    uint256 nullifierHash,
    uint256[8] calldata proof,
    uint256 proofTime
) external payable override;
```

The contract verifies the proof's validity with the worldIdRouter and updates the (nullifierHashes and addressVerifiedUntil) mappings to reflect the user's verification status and validity period.

4.2 WLDVault

The WLDVault contract is a savings vault designed to allow verified users within the Worldcoin ecosystem to deposit their WLD tokens and earn yield over time. The WLDVault contract's functionality include:

- **Deposits**: Users can deposit WLD tokens into the vault. Upon deposit, the tokens start accruing interest immediately if the user is verified. The contract maintains a record of each user's deposited amount.
- **Withdrawals**: Users can withdraw their deposited tokens along with the accrued interest at any time. The contract ensures that the correct amount of interest is calculated and distributed upon withdrawal.
- **Interest Calculation**: Interest is computed based on a predefined yield rate, the amount deposited, and the duration for which the tokens have been saved in the vault.

4.2.1 Deposits and Withdrawals

Users primarily engage with the WLDVault contract through its deposit and withdrawal functions. These functionalities allow users to deposit their WLD tokens and subsequently withdraw them along with any accrued interest. The accrued interest is calculated and immediately added to the user's balance during each deposit operation.

The deposit(...) function allows users to deposit WLD tokens into the vault. This function can be called with or without specifying an account, allowing deposits for other users.

During the deposits, the vault interacts with WorldIDAddressBook to query the addressVerifiedUntil mapping to obtain a timestamp, which will be saved to the user's deposit struct Deposit as endTime variable. This timestamp refers to the point to which the user can accrue interest for his deposited amount.

```
function deposit(uint256 amount) external virtual;
```

The `withdraw(...)` and `withdrawAll(...)` functions allow users to withdraw their tokens from the vault. Users can also use the `withdrawWithSig(...)` function to authorize withdrawals with an off-chain signature.

```
function withdraw(uint256 amount) external virtual;

function withdrawAll() external virtual;

function withdrawWithSig(
    address receiver,
    uint256 amount,
    uint256 nonce,
    bytes calldata signature
) external virtual;

```

Moreover, users can use the `recoverDeposit()` function to withdraw their deposited amounts only, excluding the accrued interests.

```
function recoverDeposit() external virtual;
```

4.2.2 Interest Calculation

Interest in the `WLDVault` contract is calculated based on several key parameters:

- **Yield Rate**: The annual percentage yield for the vault. This value is set by the owners of the `WLDVault` contract.
- **Deposit Amount**: The amount of `WLD` tokens deposited by the user.
- **Time Elapsed**: The time (in seconds) that has elapsed since the last interest calculation.

The formula for the interest calculation is as follows:

$$\text{Interest} = \left(\frac{\text{Deposit Amount} \times \text{Yield Rate} \times \text{Time Elapsed}}{\text{Seconds Per Year} \times 10000} \right)$$

Additionally, users' interests are controlled by additional parameters managed by the contract owner:

1. `maxYieldAmount`: The maximum amount of tokens for which the user is able to earn interest.
2. `yieldAccrualDeadline`: The maximum timestamp to which the user is able to accrue rewards.

5 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.

6 Issues

6.1 [Low] Incorrect check in `_withdraw(...)` function could prevent valid withdrawals

File(s): `WLDVault.sol`

Description: The `_withdraw(...)` function is an internal function responsible for processing user withdrawal requests. It performs several validation checks before updating all states and executing the withdrawal. The first validation check, `contractBalance < amount`, ensures that the contract's balance has enough tokens to cover the withdrawal. The second validation check, `contractBalance - amount < totalUserDeposits`, aims to prevent withdrawals that would dip into other users' deposits. However, this second check makes an incorrect assumption about the `totalUserDeposits` variable, as it includes the current user deposits without subtracting the withdrawn amount.

Below is the relevant section of the code:

```
function _withdraw(address user, uint256 amount) internal {
    // ...
    uint256 contractBalance = token.balanceOf(address(this));
    if (
        contractBalance < amount ||
        // @audit: `totalUserDeposits` includes the current user withdrawn `amount`
        contractBalance - amount < totalUserDeposits
    ) {
        revert ContractInsolvent();
    }
    // ...
}
```

For example:

- User 1 deposits 100 tokens;
- User 2 deposits 200 tokens;
- `totalUserDeposits` is 300 tokens;

After some time, User 1 earned 10 tokens, and User 2 earned 20 tokens in interest. The `contractBalance` must now be at least 330 tokens (110 + 220) while `totalUserDeposits` remains at 300 since it does not account for accrued interest. If User 1 attempts to withdraw 50 tokens, the check will fail:

```
contractBalance - amount < totalUserDeposits
// 330 - 50 < 300
```

This issue will be amplified when the contract doesn't own enough reward tokens.

Recommendation(s): Consider updating `totalUserDeposits` before the check to account for the current withdrawal.

Status: Fixed

Update from the client: superseded by 0eb0dac952a005a88c92c73a6e392aa60543cc32

Update from Nethermind Security: Issue fixed. For a more complete condition here, you could also consider verifying if the yield source balance covers the `fromYieldSourceAmount`. However, implementing this check would increase gas costs of the function.

6.2 [Low] Potential interest loss due to Address Book and verification length changes

File(s): `WLDVault.sol`

Description: The `WorldIDAddressBook` contains a mapping that `WLDVault` queries to determine the verification timestamp until which a user can accrue rewards. This timestamp is stored in the user's `Deposit` structure as the `endTime` variable during the call to the `deposit(...)` function. Once a user deposits, the `endTime` should only increase by depositing additional tokens or calling the `refresh(...)` function if the user is verified for additional periods. In both cases, the `WorldIDAddressBook` is queried again to check if the verification timestamp has been extended.

However, an issue arises when the `addressBook` in `WLDVault` is updated to a new deployment of the `WorldIDAddressBook` contract. The new address book will contain a fresh mapping, causing any previously verified users to become unverified. If a user then calls `deposit(...)`, it will reset their `endTime` to zero, causing the user to stop accruing rewards. This can result in users losing a portion of their rewards if this occurs before their original `endTime` expires.

A similar situation occurs when the `verificationLength` is updated to a shorter value, as re-verifying a user can set an earlier `endTime`. Consequently, users will accrue fewer rewards if they interact with the protocol before reaching their original `endTime`.

Recommendation(s): To avoid potential inconsistencies in rewards distribution, consider disallowing the user's `endTime` to be decreased within the `WLDVault` contract.

Status: Fixed

Update from the client: Addressed in db0044c03b8ae983449464e2f3373632f4cedd36.

6.3 [Info] Payable `verify(...)` function may lead to locked Ether

File(s): `WorldIDAddressBook.sol`

Description: The `verify(...)` function in the `WorldIDAddressBook` contract is currently marked as `payable`, allowing Ether to be sent during invocation. However, this function calls `verifyProof(...)` from the `worldIdRouter` contract without utilizing any Ether in the process.

As there is no mechanism to withdraw Ether from the `WorldIDAddressBook` contract, any Ether sent to this function will become permanently locked in the contract.

Recommendation(s): To address this issue, consider removing the `payable` modifier from the `verify(...)` function. If the `payable` modifier is necessary for interface compatibility, implement a mechanism to prevent users from sending Ether during the function call.

Status: Acknowledged

Update from the client: Acknowledged, won't fix.

6.4 [Best Practice] Avoid the usage of magic numbers

File(s): WLDVault.sol

Description: A magic number is a numerical value hard-coded in the source code, which, without the context, makes it hard to understand the developer's intent in using that number. To avoid this anti-pattern, it's recommended to define a constant with a meaningful name instead of using magic numbers directly.

An example of this behavior is the numeric literal `10000` used in the `calculateInterest(...)` function.

Recommendation(s): To keep the code easier to read and comprehend, consider defining a constant like `ONE_HUNDRED_PERCENT` instead of using the literal `10000`.

Status: Fixed

Update from the client: Addressed in 3b729f38444f848e1a379c22c5da3731711e4a9b.

Update from Nethermind Security: Issue fixed. Note that this comment can be completed in a similar way to the yieldRate, indicating the 2 decimals representation, i.e basis points.

6.5 [Best Practices] Missing inputs validation

File(s): WLDVault.sol, WorldIDAddressBook.sol

Description: Whenever the caller is expected to provide input to the function, performing validation checks on the values provided is recommended as a best practice. It ensures that the contract does not end up in an undesired state. Below is a non-exclusive list of significant missing checks:

- WorldIDAddressBook -> verificationLength: The constructor ensures verificationLength cannot be set to 0, but this validation is absent in the setVerificationLength(...) function. This inconsistency could result in verificationLength being set to 0, preventing user validation;
- The deposit(...) function does not check if the account is not the zero address. As a result, a user might mistakenly lose the funds by sending them to the zero address;
- The token and addressBook variables in the WLDVault contract lack zero address check in both the contract constructor and the setAddressBook(...) function;
- The maxProofTime can be set to 0 in the setMaxProofTime(...) function;
- The yieldRate and maxYieldAmount variables could be bounded to reasonable values to prevent setting them unreasonably high by accident;

Recommendation(s): To prevent the contracts from entering an unexpected state, consider extending the input validation to eliminate the risks mentioned above from happening.

Status: Acknowledged

Update from the client: Partially addressed in f9fea2ac999c591b90df9e47aae9caac7904e7b4. Acknowledged missing deposit check for gas savings.

Update from Nethermind Security: The outlined checks are fixed except for the acknowledged deposit check. We have added two additional bullet points to the list where the checks could be implemented.

Update from the client: Partially addressed in 28171872a23a03bef49898eb939934765157fa4b. Bounding the yieldRate and maxYieldAmount is not considered neccesary.

6.6 [Best Practices] Redundant check in refresh(...) function

File(s): WLDVault.sol

Description: The refreshInterest(...) function checks the current user's amount and exits if it is zero:

```
function refreshInterest(...) internal {
    if (userDeposit.amount == 0) return;
    // ...
}
```

However, this internal function is never invoked when userDeposit.amount == 0 as this check is already performed in all external functions that call it. Specifically, in the public refresh(...) function, the check is implemented as follows:

```
function refresh(...) public override {
    // ...
    Deposit storage userDeposit = getDeposit[account];
    // @audit Duplicated check
    if (userDeposit.amount != 0) refreshInterest(userDeposit);
    // ...
}
```

Recommendation(s): Remove the redundant check from the public refresh(...) function.

Status: Fixed

Update from the client: Addressed in a525fb9f805eb59d3c6c511918f477eaef4c8bc5.

6.7 [Best Practices] The WalletVerified event is not emitted in the verify(...) function

File(s): WorldIDAddressBook.sol

Description: The WorldIDAddressBook contract defines the WalletVerified event to be emitted upon account verification. However, the verify(...) function, which handles the account verification process, currently does not emit this event upon successful verifications.

Recommendation(s): Consider implementing the emission of the WalletVerified event within the verify(...) function. Alternatively, if the event is unnecessary, consider removing its definition to maintain clarity and prevent confusion.

Status: Fixed

Update from the client: Event removed in ba60f3355c34b55f441067dd2e9e654bcad800dc.

6.8 [Best Practices] The `verify(...)` function does not follow C-E-I pattern

File(s): `WorldIDAddressBook.sol`

Description: The Checks, Effects, and Interactions pattern is generally a best practice that functions should follow. It prevents the possible impact of reentrancy as the state of the contract is updated before the external interactions.

The `verify(...)` function inside the `WorldIDAddressBook` contract does not follow this pattern. It invokes the `verifyProof(...)` function on the `worldIdRouter` before updating the state of mappings for address verification. Fortunately, there is currently no direct impact as `worldIdRouter` is a contract controlled by Worldcoin.

Recommendation(s): To prevent future issues, consider following this pattern by updating the state of the contract before external calls.

Status: Fixed

Update from the client: Addressed in 446c89ec08fec6fbad9259eae5bd9a1f5ee12e9b.

7 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.

Remarks about Worldcoin documentation

The **Worldcoin** team has provided documentation about their protocol in the form of in-line comments within the code. However, these comments did not provide a comprehensive explanation of the architecture and decisions behind various functionalities. Despite this, the Worldcoin team was available to address any questions or concerns from the Nethermind Security team.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge build
[] Compiling...
[] Compiling 1 files with Solc 0.8.24
[] Solc 0.8.24 finished in 629.46ms
Compiler run successful!
```

8.2 Tests Output

```
> forge test

Ran 32 tests for src/test/WLDVault.t.sol:WLDVaultTest
[PASS] testCanDepositForUnverifiedUser() (gas: 149571)
[PASS] testCanDepositIfUnverified() (gas: 146492)
[PASS] testCanRefreshForOtherUser() (gas: 51915)
[PASS] testCanRefreshWithoutDeposit() (gas: 42371)
[PASS] testCannotDepositZeroTokens() (gas: 11124)
[PASS] testCannotDepositZeroTokensForOtherUser() (gas: 13379)
[PASS] testCannotRecoverDepositIfNoDeposit() (gas: 13217)
[PASS] testCannotRenounceOwnership() (gas: 17115)
[PASS] testCannotWithdrawIfNoDeposit() (gas: 13252)
[PASS] testCannotWithdrawLessThanBalance() (gas: 158730)
[PASS] testContractInsolvent() (gas: 147119)
[PASS] testDepositForOtherUser() (gas: 197496)
[PASS] testDepositForOtherUserEarnsInterest() (gas: 167940)
[PASS] testDepositForOtherUserIncreasesExistingDeposit() (gas: 187530)
[PASS] testDepositForOtherUserRespectsCap() (gas: 168048)
[PASS] testDepositYieldIsCapped() (gas: 153355)
[PASS] testInterestIsPausedWhenUnverified() (gas: 178741)
[PASS] testOwnerCanSetYieldRate() (gas: 29792)
[PASS] testOwnerCanUpdateAddressBook() (gas: 29534)
[PASS] testOwnerCanUpdateMaxYieldAmount() (gas: 24422)
[PASS] testOwnerCanUpdateYieldAccrualDeadline(uint256) (runs: 256, : 46124, ~: 46124)
[PASS] testRecoverDeposit() (gas: 135963)
[PASS] testSimpleFlow(uint256,uint256) (runs: 256, : 156986, ~: 156988)
[PASS] testUserCanWithdrawAtAnyTime(uint8) (runs: 256, : 131211, ~: 131211)
[PASS] testUserDepositsTracking() (gas: 180171)
[PASS] testUsersCanIncreaseTheirDeposit() (gas: 176644)
[PASS] testWithdrawAll() (gas: 144820)
[PASS] testWithdrawWithSig(uint256,uint256,uint256) (runs: 256, : 159782, ~: 159791)
[PASS] testWithdrawWithSigInvalidSignature(uint256,uint256,uint256) (runs: 256, : 167468, ~: 167468)
[PASS] testWithdrawWithSigNonceReuse(uint256,uint256,uint256) (runs: 256, : 202324, ~: 202324)
[PASS] testYieldAccrualDeadlineAffectsInterestCalculation() (gas: 182677)
[PASS] testYieldAccrualDeadlineDoesNotAffectPrincipal() (gas: 150668)
Suite result: ok. 32 passed; 0 failed; 0 skipped; finished in 821.07ms (899.18ms CPU time)
```

```
Ran 13 tests for src/test/WorldIDAddressBook.t.sol:WorldIDAddressBookTest
[PASS] testCanReverifyAnytime(uint256) (runs: 256, : 112637, ~: 112637)
[PASS] testCanSwitchVerificationAddressWhenItExpires(address) (runs: 256, : 132122, ~: 132108)
[PASS] testCannotRenounceOwnership() (gas: 16404)
[PASS] testCannotReverifyDifferentAddressBeforeExpiry(address,uint256) (runs: 256, : 128802, ~: 128802)
[PASS] testCannotVerifyWithAProofInTheFuture() (gas: 60110)
[PASS] testCannotVerifyWithInvalidProof() (gas: 49811)
[PASS] testConstructorVerifiesArguments() (gas: 751183)
[PASS] testOwnerCanSetVerificationLength() (gas: 23767)
[PASS] testOwnerCanUpdateGroupId() (gas: 23865)
[PASS] testOwnerCanUpdateMaxProofTime() (gas: 23712)
[PASS] testOwnerCanUpdateRouterAddress() (gas: 30097)
[PASS] testUserCanGetVerified() (gas: 93689)
[PASS] testUserCannotGetVerifiedWithOldProof() (gas: 62756)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 821.15ms (1.42s CPU time)

Ran 2 test suites in 850.55ms (1.64s CPU time): 45 tests passed, 0 failed, 0 skipped (45 total tests)
```

Remarks about Worldcoin test suite

The Worldcoin team has developed an broad testing suite that comprehensively covers a variety of user workflows, including scenarios related to interest calculation. However, it is worth noting that initially, the vault was pre-funded with a substantial number of tokens. This approach resulted in certain states of the contract unexplored during testing phases.

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 procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, 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