Security Review Report
NM-0226 Worldcoin

NETHERMIND
SECURITY

(April 29, 2024)

Contents

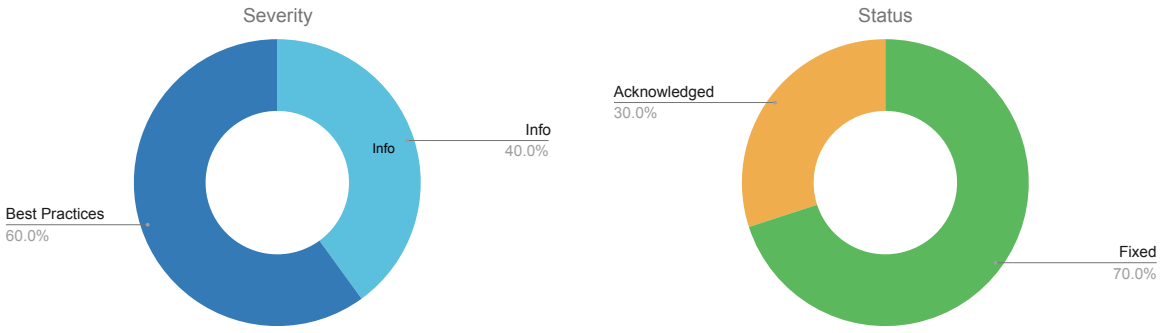
1 Executive Summary

This document presents the security review conducted by Nethermind Security for the Savings contract developed by Worldcoin. The new functionality enables the World App users to earn interest on their stablecoins. The users can deposit `USDC.e` or `USDC` tokens into the `Savings` contract and receive the *savings DAI* token (sDAI) in return. Users can redeem their initial deposit with the accrued interest at any time. The `Savings` contract will be deployed on the Optimism network. The conversion between the stablecoins and the `sDAI` token will use the rate returned from the *DAI Savings Rate* Oracle contract deployed on Optimism by MakerDAO.

Initially, the Worldcoin team plans to pre-fund about a year's worth of interest for the users. This involves the initial deposit of funds into the `Savings` contract, including the `USDC.e` and bridged `sDAI` tokens. The contract defines two different fees: The withdrawal fee, which is applied for each redemption operation and computed as a percentage of the withdrawn amount, and the performance fee, which is applied to the accrued interest for each user. Currently, the performance fee is hard-coded as an input to the function, and the logic handling its computation is not yet implemented.

The audited code comprises 115 lines of code in Solidity. The `Worldcoin` team has provided the necessary explanations to assist the audit. The audit was performed using: (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts. **Along this document, we report** 10 points of attention, classified as `Informational` or `Best Practices`. 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** (0), **Undetermined** (0), **Informational** (4), **Best Practices** (6). **(b) Distribution of status: Fixed** (7), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Apr 22, 2024
Final Report	Apr 29, 2024
Methods	Manual Review, Automated Analysis
Repository	helper-contracts
Commit Hash	c48a2953ae2ccd5661f996a73d17acd27c1e780b
Final Commit Hash	4001cdd06832c08494784c9de989c7b49886febe
Documentation	PR description
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/Savings.sol	115	78	67.8%	49	242
	Total	**115**	**78**	**67.8%**	**49**	**242**

3 Summary of Issues

	Finding	Severity	Update
1	Possible phishing attack due to the use of `tx.origin` in the `onlyRelayer(...)` modifier	Info	Acknowledged
2	Possible underflow within redeem functions	Info	Fixed
3	The `onlyRelayer(...)` modifier reverts with an incorrect relayer address	Info	Fixed
4	Withdrawal fee can be bypassed for small `sDAI` amounts	Info	Acknowledged
5	A frequently used number literal `10_000` could be made `constant`	Best Practices	Fixed
6	Missing input validations	Best Practices	Fixed
7	Perform input validation before external calls	Best Practices	Fixed
8	The `MIN` and `MAX` DSR conversion rate state variables could be made `constant`	Best Practices	Fixed
9	The `validateConversionRate(...)` function name contains a typo	Best Practices	Fixed
10	Unchecked return values of `transfer(...)` and `transferFrom(...)` calls	Best Practices	Acknowledged

4 System Overview

The Savings contract offers the World App users the ability to earn interest on their stablecoins. Users can deposit USDC and USDC.e tokens into the contract and receive the savings DAI token in return. The sDAI token can be redeemed back for stablecoins at any point in time. The token exchange rate, applied on both the deposit and withdrawal operations, is queried from the *Dai Savings Rate* Oracle (DSR) deployed on Optimism by MakerDAO. In the code, the exchange rate is referred to as the *conversion rate*.

To simplify the user experience, a special entity called Relayer interacts with the Savings contract on behalf of the user. The user is expected to sign a single approval transaction using his Safe smart contract wallet (formerly Gnosis Safe). He specifies the action to be performed and the token to be used: sDAI for withdrawals and USDC or USDC.e for deposits. Using the user's signature, the relayer can call the Safe contract and approve the Savings contract on the user's behalf. In the second step, the relayer can call the appropriate deposit or withdrawal function, which will transfer the tokens from the user to the Savings contract. For efficiency reasons, both calls executed by the relayer will be batched together in a single Multicall3 transaction.

4.1 Actors of the system

The system distinguishes between 3 main roles:

- **Owner Role**: The Savings contract owner is a multi-signature wallet owned by the Worldcoin team, allowed to set and update protocol parameters. Below are the main functions an owner can call:
  - setWithdrawalFee(...): This function allows to set the withdrawal fee, expressed in basis points (bips), applied to the sDAI amount during redemption operations.
  - setFeeRecipient(...): This function sets the address of the account that will receive the withdrawal fees during the redemption operation.
  - setRelayerAuthorized(...): This function enables adding or removing addresses from the mapping of authorized Relayers.
  - withdraw(...): This function facilitates tokens withdrawal from the Savings contract to a defined recipient.
- **Relayer Role**: Relayers are Externally Owned Accounts (EOAs) that are whitelisted within the contract and allowed to execute the different deposit and redemption operations.
- **End users**: World App users are only required to sign the ERC20 approval transaction prior to the deposit or redemption operation. Every user owns a Safe wallet (formerly Gnosis Safe) and doesn't have to interact directly with the Savings contract.

4.2 Deposits

The Savings contract will be initially funded by the Worldcoin team, enabling users to deposit USDC and USDC.e into the contract to earn interest on their stablecoins. For that, the contract provides the following functions:

- depositUSDC(...): This function allows users to deposit USDC into the contract and receive an amount of sDAI in return, based on the exchange rate received from the MakerDAO DSR Oracle. This operation is performed by the Relayer on behalf of the user.
- depositUSDCE(...): Similarly, this function allows users to deposit USDC.e and receive sDAI in exchange.

To make a deposit, users must approve the USDC or USDC.e token transfer beforehand. For that, they need to sign the approval transaction using their Safe account. This signed transaction is then aggregated in a multi-call, followed by the deposit function call. Subsequently, the Relayer executes the two actions in a single transaction using the Multicall3 contract.

4.3 Withdrawals

Users have the option to redeem either USDC or USDC.e by providing an equivalent amount of sDAI. The Savings contract offers the following redemption functions:

- redeemUSDC(...): This function allows users to exchange sDAI for USDC. The process involves two different fees: Initially, a withdrawal fee is applied as a percentage of the withdrawn amount, followed by a performance fee imposed on the accrued interest. The withdrawal operation is performed by the Relayer on behalf of the user.
- redeemUSDCE(...): Similarly, users can redeem USDC.e using this function. The process is similar to the redeemUSDC(...) function.

Similarly to deposits, in order to initiate a redemption, users are required to sign the sDAI ERC20 approval transaction using their Safe account. A multi-call aggregating this approval with the redeem function call is then executed by the Relayer.

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 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 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 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 [Info] Possible phishing attack due to the use of `tx.origin` in the `onlyRelayer(...)` modifier

File(s): `Savings.sol`

Description: The `onlyRelayer(...)` modifier is used to restrict access to contract functions authorized only to EOAs identified as relayers. This modifier uses the `tx.origin` global variable to authorize the caller to an operation. However, The use of `tx.origin` is associated with possible risks of `phishing attacks` and its use is not recommended for authorization by the best practices indicated in the Solidity language documentation. Also, there is a chance that `tx.origin` will be removed from the Ethereum protocol in the future, so code that uses `tx.origin` won't be compatible with future releases Vitalik Buterin: 'Do NOT assume that tx.origin will continue to be usable or meaningful'.

```
modifier onlyRelayer() {
    // @audit tx.origin used for authorization
    if (!isAuthorizedRelayer[tx.origin]) {
        revert UnauthorizedRelayer(msg.sender);
        _;
    }
}
```

`tx.origin` is a global variable in Solidity that returns the account address that initiated the transaction. Using this variable for authorization could make a contract vulnerable if an authorized account calls into a malicious contract, so in a scenario where an attacker successfully persuades an authorized relayer to initiate a transaction by calling a function in their malicious contract and subsequently redirects the transaction to a function in the `Savings` contract that uses this modifier, the attacker could circumvent the authorization check. This is because the address of the attacking contract would be in `msg.sender`, while the address of the authorized relayer, who initially initiated the transaction, would be in `tx.origin`, thereby enabling the attacker to bypass the validation.

Given the level of authority that relayers possess, an attacker could execute calls to the functions `unwrap(...)`, `depositUSDC(...)`, `depositUSDCE(...)`, `redeemUSDC(...)` and `redeemUSDCE(.. .)`, but since these functions involve only an exchange of tokens between the contract and a receiving account there is no major impact for the contract.

Recommendation(s): Consider revisiting the system design and avoid relying on `tx.origin` for authorization. A recommended solution is implementing a signature validation scheme to verify the signatures of authorized relayers. Utilizing the OpenZeppelin ECDSA library for this purpose is recommended, along with adhering to the guidelines outlined in the ERC-712 standard.

Status: Acknowledged

Update from the client: We acknowledge the risk of relying on tx.origin for authentication. To minimize the overhead on our backend, we will not implement ECDSA signature verification as a replacement for the tx.origin check.

6.2 [Info] Possible underflow within redeem functions

File(s): Savings.sol

Description: The functions `redeemUSDC(...)` and `redeemUSDCE(...)` enable users to redeem `USDC` or `USDCE`, respectively, by providing `sDAI`. During redemption, two types of fees are involved: a performance fee, which is specified as a parameter, and a withdrawal fee, calculated as a percentage of the withdrawn amount. However, the current implementation only checks whether the withdrawn `amount` is greater than the performance fee, while both fees are subtracted from that amount. This can result in an underflow error during the computation process.

```
function redeemUSDCE(address recipient, uint256 amount, uint256 performanceFee) external onlyRelayer {
    SDAI.transferFrom(recipient, address(this), amount);

    require(amount > 0, "Redeem amount is 0");
    // @audit The check only covers the `performanceFee`
    require(performanceFee < amount, "Performance fee is greater than redeem amount");

    uint256 withdrawalFee = (amount * withdrawalFeeBips) / 10_000;

    uint256 amountAfterWithdrawalFee = amount - withdrawalFee;

    // @audit Possible underflow error if
    // `amountAfterWithdrawalFee < performanceFee`
    uint256 amountAfterPerformanceFee = amountAfterWithdrawalFee - performanceFee;
    require(amountAfterPerformanceFee > 0, "Amount after fees is 0");
    // ...
}
```

Recommendation(s): Consider verifying whether the withdrawn `amount` is greater than the sum of the performance and withdrawal fees.

Status: Fixed

Update from the client: Fix: b90eeb7f566c34f9dcee06a0cfc5a234202f5704

6.3 [Info] The `onlyRelayer(...)` modifier reverts with an incorrect relayer address

File(s): Savings.sol

Description: The `onlyRelayer(...)` modifier ensures that the deposit/redeem multi-call transaction was initiated by an authorized relayer. The modifier checks that the `tx.origin` is present in the `isAuthorizedRelayer` mapping, and if not, the `UnauthorizedRelayer` error is thrown. However, the thrown error uses the address of the `msg.sender`, which corresponds to the multi-call contract, instead of the `tx.origin` relayer address that initiated the transaction.

```
modifier onlyRelayer() {
    if (!isAuthorizedRelayer[tx.origin]) {
        // @audit The msg.sender is the multi-call contract, not the relayer.
        revert UnauthorizedRelayer(msg.sender);
    }
    _;
}
```

Recommendation(s): To keep the condition consistent with the error value, consider reverting with the `tx.origin` address that initiated the transaction instead of the direct caller of the function.

Status: Fixed

Update from the client: Fix: 0fea25245b3f8337d0254af18985c2d3c75bc0a0

6.4 [Info] Withdrawal fee can be bypassed for small sDAI amounts

File(s): Savings.sol

Description: During the redemption process, a withdrawal fee is imposed as a percentage of the withdrawn amount. However, the function doesn't validate whether the resulting fee amount is non-zero. This allows users to manipulate the withdrawal amount by providing small sums, leading to the fee amount rounding down to zero, thereby bypassing the fee. To illustrate, for a fee percentage of `P`, any withdrawal amount `A` such that: `A * P < 10_000` results in zero fees.

```
function redeemUSDCE(address recipient, uint256 amount, uint256 performanceFee) external onlyRelayer {
    SDAI.transferFrom(recipient, address(this), amount);

    require(amount > 0, "Redeem amount is 0");
    require(performanceFee < amount, "Performance fee is greater than redeem amount");

   // @audit The `withdrawalFee` can be zero.
    uint256 withdrawalFee = (amount * withdrawalFeeBips) / 10_000;

   // ...
}
```

Given that fees are deducted in `sDAI`, which has 18 decimals, the necessary amounts to perform this manipulation are extremely small and don't make an important impact.

Recommendation(s): Consider incorporating a validation check in both `redeemUSDC(...)` and `redeemUSDCE(...)` functions to ensure that the computed withdrawal fee is non-zero.

Status: Acknowledged

Update from the client: We acknowledge the risk but have decided to leave as is. We also have checks on the backend which prevent the creation of a large amount of small transactions from a single user.

6.5 [Best Practices] A frequently used number literal 10_000 could be made constant

File(s): Savings.sol

Description: The `Savings` contract calculates the withdrawal fee as a percentage of the requested withdrawal amount using basis points. The integer literal `10_000` is used in multiple places throughout the codebase to serve as a value of `100` percent. Where applicable, it is considered a best practice to use named constants instead of number literals to improve the codebase's readability and give more context to the reader.

Recommendation(s): Consider refactoring the code to use a named constant with a meaningful name like `ONE_HUNDRED_PERCENT`.

Status: Fixed

Update from the client: Fix: a67820cc89e3cdb2861fdcf23243cc0a3e27a1e6

6.6 [Best Practices] Missing input validation

File(s): Savings.sol

Description: Some parts of the code do not implement proper input validation checks. To mitigate human errors, it is considered a best practice to perform input validation to ensure that the contract is always in a proper state.

The following functions do not perform the `address(0)` check, on the address parameters:

- In the `Savings` contract's `constructor(...)` the address parameters are not checked;
- In the `setFeeRecipient(...)` function the parameter `_feeRecipient` is not checked;
- In the `setRelayerAuthorized(...)` function the parameter `relayer` is not checked;
- In the `withdraw(...)` function the parameter `recipient` is not checked;

Recommendation(s): Consider adding validations for input parameters according to the recommendations above.

Status: Fixed

Update from the client: Fix: c516e21b5e764acda15c8e743168f36b03817cfd and 4001cdd06832c08494784c9de989c7b49886febe

6.7 [Best Practices] Perform input validation before external calls

File(s): Savings.sol

Description: In the functions `redeemUSDC(...)` and `redeemUSDCE(...)`, external calls are made before validating the input parameters. This may lead to unnecessary gas expenditure in cases where the transaction reverts due to input validation. For example, in the `redeemUSDC(...)` function:

```
function redeemUSDC(address recipient, uint256 amount, uint256 performanceFee) external onlyRelayer {
    // @audit External call
    SDAI.transferFrom(recipient, address(this), amount);

    // @audit Input parameter validation
    require(amount > 0, "Redeem amount is 0");
    require(performanceFee < amount, "Performance fee is greater than redeem amount");
    // ...
}
```

Recommendation(s): Consider verifying input parameters before initiating external calls. This is considered a best practice that helps save gas in the process.

Status: Fixed

Update from the client: Fix: 7b8e503be5f75de9ed5e2c14afc39cbe02c41ea8

6.8 [Best Practices] The `MIN` and `MAX` DSR conversion rate state variables could be made `constant`

File(s): Savings.sol

Description: The `Savings` contract uses the `MIN_DSR_CONVERSION_RATE` and `MAX_DSR_CONVERSION_RATE` state variables to validate the conversion rate returned by the DSR oracle. Since these variables are fixed at compile time, their mutability could be changed from `immutable` to `constant`.

Recommendation(s): Consider changing the mutability of the above-mentioned variables from `immutable` to `constant`.

Status: Fixed

Update from the client: Fix: cab8f0b08f15bb8ea7393cf7014c79036a08d5dd

6.9 [Best Practices] The `validateConversationRate(...)` function name contains a typo

File(s): Savings.sol

Description: The `validateConversationRate(...)` function validates the conversion rate returned by the DSR oracle. Given its purpose, we believe that the current function name contains a typo. Instead of `conversation`, it should contain the word `conversion`.

Recommendation(s): Consider changing the function's name from `validateConversationRate(...)` to `validateConversionRate(...)`.

Status: Fixed

Update from the client: Fix: 2f54eb710f70b91d89f4d31818a2624f4e32eeb1

6.10 [Best Practices] Unchecked return values of `transfer(...)` and `transferFrom(...)` calls

File(s): Savings.sol

Description: The relayer functions within the `Savings` contract allow the transfer of supported ERC20 tokens through swaps or withdrawals. However, these functions ignore the return values from the `transfer(...)` and `transferFrom(...)` ERC20 functions. Some tokens use this value to indicate the transfer status, returning `false` in case of failure.

While all the currently transferred tokens: `sDAI`, `USDC`, and `USDC.e` conform to the intended behavior of reverting on failure, the `withdraw(...)` function allows the withdrawal of any ERC20 token from the contract, potentially including tokens that return `false` on failure instead of reverting. Although there's no significant impact currently, this may mislead users or third parties relying on the transaction's success if the transfer actually fails despite the transaction being successful.

Recommendation(s): Consider using `safeTransfer(...)` and `safeTransferFrom(...)` functions from OpenZeppelin `SafeERC20` contract. Alternatively, implement a check to verify that the return values for `transfer(...)` and `transferFrom(...)` are `true`.

Status: Acknowledged

Update from the client: We acknowledge the risk but have decided to leave as is given that we only expect to receive USDC.e, USDC, and sDAI which all conform to the ERC20 standard and revert on fail.

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 contracts documentation is presented through inline comments within the code, providing explanations and insights about their implementation. Additionally, the PR description accessible at this link, outlines the important funtionnalities implemented within the contract. Furthermore, the `Worldcoin` team was available to address any inquiries or concerns raised by the Nethermind Security team.

8 Test Suite Evaluation

8.1 Contracts Compilation

```
forge build
[] Compiling...
[] Compiling 23 files with 0.8.24
[] Solc 0.8.24 finished in 3.44s
Compiler run successful with warnings:
Warning (2519): This declaration shadows an existing declaration.
   --> test/Savings.t.sol:238:9:
    |
238 |         uint256 conversionRate = 950_000_000_000_000_000_000_000_000;
    |         ^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/Savings.t.sol:56:5:
   |
56 |     uint256 conversionRate = 1_050_000_000_000_000_000_000_000_000; // Represents 1.05 in fixed-point format
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> test/Savings.t.sol:259:9:
    |
259 |         uint256 conversionRate = 2_950_000_000_000_000_000_000_000_000;
    |         ^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> test/Savings.t.sol:56:5:
   |
56 |     uint256 conversionRate = 1_050_000_000_000_000_000_000_000_000; // Represents 1.05 in fixed-point format
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

8.2 Tests Output

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

Ran 12 tests for test/Savings.t.sol:SavingsTest
[PASS] testConvertUSDCEtoSDAI() (gas: 171653)
[PASS] testConvertUSDCtoSDAI() (gas: 171672)
[PASS] testInvalidFee() (gas: 10548)
[PASS] testOwnerWithdrawal() (gas: 192080)
[PASS] testRedeemUSDCENoPerformanceFee() (gas: 217986)
[PASS] testRedeemUSDCNoPerformanceFee() (gas: 217998)
[PASS] testRedeemUSDCWithPerformanceFee() (gas: 217967)
[PASS] testRevertOnTooHighConversionRate() (gas: 153950)
[PASS] testRevertOnTooLowConversionRate() (gas: 153908)
[PASS] testRevertRedeemWithoutApprovalFunction() (gas: 146648)
[PASS] testUSDCEUnwrap() (gas: 172977)
[PASS] testUSDCEUnwrapRevertsWhenNoUSDCMinted() (gas: 108407)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 4.99ms (2.42ms CPU time)

Ran 6 tests for test/BatchSafeAllowanceModuleTransfers.t.sol:BatchSafeAllowanceModuleTransfersTest
[PASS] testAllowanceResetTimeMinHasNotYetTranscurredReverts(uint8) (runs: 1009, : 1202223, ~: 617340)
[PASS] testBatchOwnableUnathourizedAccountReverts(address) (runs: 1015, : 27908, ~: 27908)
[PASS] testBatchSucceeds(uint8,uint16) (runs: 1009, : 1622688, ~: 729497)
[PASS] testBatcherDoesNotHaveEnoughAllowanceReverts() (gas: 515705)
[PASS] testBatcherIsNotAllowanceModuleDelegateReverts(address) (runs: 1015, : 104621, ~: 104621)
[PASS] testEnableModuleSucceeded() (gas: 14945)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 1.98s (3.32s CPU time)

Ran 2 test suites in 1.99s (1.98s CPU time): 18 tests passed, 0 failed, 0 skipped (18 total tests)
```

9 About Nethermind

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

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

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

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

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

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

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

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

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

Learn more about us at nethermind.io.

General Advisory to Clients

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

Disclaimer