Security Review Report
NM-0472 Renzo Bridge

NETHERMIND
SECURITY

(May 1, 2025)

Contents

1 Executive Summary

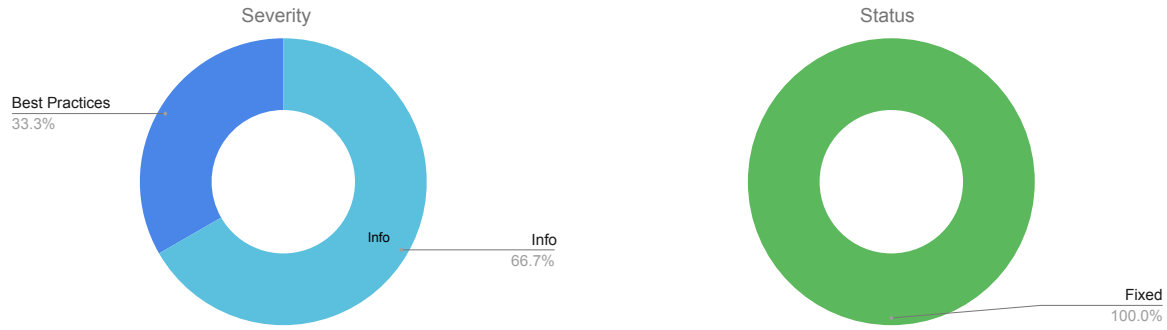
This document presents the security review conducted by Nethermind Security for the bridge smart contracts of Renzo Protocol. Renzo is a restaking solution that abstracts and manages AVS strategies for Liquid Restaking Tokens (LRTs), simplifying Ethereum restaking and making it accessible to a broader audience. By leveraging EigenLayer, Renzo enables users to restake their ETH without the complexities of node operation, AVS strategy selection, or direct smart contract interactions.

The security review focused on the smart contracts designed to enable deposits into the core Renzo protocol from Layer 2 networks. In return for making deposits, users receive `xezETH` tokens, which serve as the Layer 2 equivalent of `ezETH`.

The audited code comprises of **896** lines of code written in the Solidity language.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report** six points of attention, where four are classified as `Informational` and two are classified as `Best Practice` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tests. Section 9 concludes the document.



(a) (b)

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (4), **Best Practices** (2). **Distribution of status: Fixed** (2), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	April 15, 2025
Final Report	May 1, 2025
Repository	Contracts
Start Commit	c2bdea417781d772a0a50b6401783ce84d068e2e
Final Commit	cfe7249d386ab916ab699be86da25db6cbc06aa3
Documentation	Docs
Documentation Assessment	High
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/Bridge/L2/xRenzoDepositNativeBridge.sol	323	222	68.7%	95	640
2	contracts/Bridge/L2/IxRenzoDeposit.sol	12	1	8.3%	3	16
3	contracts/Bridge/L2/xRenzoDepositNativeBridgeStorage.sol	25	16	64.0%	16	57
4	contracts/Bridge/L2/Oracle/RenzoOracleL2Storage.sol	6	1	16.7%	2	9
5	contracts/Bridge/L2/Oracle/RenzoOracleL2.sol	34	13	38.2%	11	58
6	contracts/Bridge/L2/Oracle/IRenzoOracleL2.sol	4	1	25.0%	1	6
7	contracts/Bridge/L2/PriceFeed/HyperlaneSender.sol	122	28	23.0%	28	178
8	contracts/Bridge/L2/PriceFeed/HyperlaneReceiver/HyperlaneReceiver.sol	68	34	50.0%	16	118
9	contracts/Bridge/.../HyperlaneReceiver/HyperlaneReceiverStorage.sol	6	3	50.0%	3	12
10	contracts/Bridge/L2/ValueTransfer/IValueTransferBridge.sol	18	1	5.6%	3	22
11	contracts/Bridge/L2/ValueTransfer/OP/LidoOPValueTransfer.sol	44	26	59.1%	13	83
12	contracts/Bridge/L2/ValueTransfer/OP/EthOPValueTransfer.sol	35	27	77.1%	10	72
13	contracts/Bridge/L2/ValueTransfer/ARB/LidoArbValueTransfer.sol	51	25	49.0%	13	89
14	contracts/Bridge/L2/ValueTransfer/ARB/EthArbValueTransfer.sol	34	27	79.4%	10	71
15	contracts/Bridge/L1/xRenzoBridgeReceiver.sol	87	44	50.6%	25	156
16	contracts/Bridge/L1/IwstETH.sol	4	9	225.0%	1	14
17	contracts/Bridge/L1/xRenzoBridgeReceiverStorage.sol	23	9	39.1%	9	41
	Total	**896**	**487**	**54.4%**	**259**	**1642**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect configuration of token time discount on Arbitrum	Info	Fixed
2	Incorrect handling of `WETH` in the `sweep(...)` function	Info	Fixed
3	Incorrect oracle configuration for `wstETH` token on Base	Info	Fixed
4	The `LidoOPValueTransfer` contract unnecessarily grants the `wstETH` allowance to the `lidoBridge` address	Info	Fixed
5	Incorrect comments	Best Practices	Fixed
6	Unused events	Best Practices	Fixed

4 System Overview

Renzo's bridging contract architecture is designed to transfer assets and price data between Layer 2 networks and Ethereum mainnet within the Renzo Protocol. Its primary goal is to allow users on the L2s to mint **ezETH** by depositing native ETH or other supported collateral tokens. Deposited assets are then bridged to the L1 for staking on Renzo. The bridge system incorporates two main components: a funds bridging mechanism that handles the asset transfers and a cross-chain price feed messaging system that ensures synchronized valuation data between chains.

4.1 Actors

The bridge system involves several distinct roles and participants:

- **L2 users**: Initiate deposits in native ETH, WETH, or wstETH tokens through `xRenzoDepositNativeBridge`. Users receive optimistically minted cross-chain `ezETH` tokens (`xezETH`), representing their claim to the underlying assets once the bridging is complete.
- **Bridge admins**: Hold responsibility for governance and maintenance across both L1 and L2 contracts. Can update parameters, including fee shares, discount levels, supported tokens, and price feed sources. They possess exclusive rights (via role-based access control) to execute recovery functions to retrieve accidentally sent tokens.
- **Oracle providers and messaging relayers**: The price oracle (Chainlink) provides real-time asset prices on L2 networks. Messaging components: `HyperlaneSender` and `HyperlaneReceiver` contracts propagate the price of ezETH from Arbitrum to other L2 networks via the Hyperlane infrastructure.

4.2 Architectural Components

4.2.1 L1 Receiver contract

The `xRenzoBridgeReceiver` contract on Ethereum mainnet receives the assets bridged from L2s. Its core responsibility is to process incoming deposits—converting, unwrapping, and depositing funds into the Renzo protocol.

- **Processing**: On receipt of tokens, the contract detects and converts these to their native forms where necessary. Any WETH balance is unwrapped into ETH, and any wstETH is unwrapped into stETH.
- **Protocol deposit**: After the conversion, the `ETH` and `stETH` tokens are deposited into Renzo Protocol's `RestakeManager` contract, triggering the mint of `ezETH`
- **Token locking and burning**: The contract then interacts with a dedicated lockbox mechanism. The newly minted `ezETH` tokens are transferred into the lockbox, and the L1 bridge contract receives `xezETH` in return, which is burned to keep L2 and L1 supplies in sync with their collateral backing.

4.2.2 L2 Deposit contract

The `xRenzoDepositNativeBridge` contract is responsible for accepting user deposits on the L2. It immediately mints `xezETH` tokens in exchange for the deposited assets while batching funds for subsequent bridging to L1.

- **User deposits**: Users initiate deposits in native ETH or supported ERC20 tokens.
- **Fee collection**: A bridge fee + an additional time-based discount fee are deducted from each deposit. The time-based fee covers the duration for which user funds do not produce yield (typically 1 week while the bridging process takes place) and represents less than 0.1 percent of the deposit. The fees are forwarded to a fee collector address.
- **Price oracle integration**: Before minting `xezETH`, the deposited asset value is measured in ETH using an oracle. This ensures that the minting process reflects an up-to-date `ezETH/ETH` exchange rate.
- **Bridging execution**: Funds that reside in the L2 contract are later "swept" and sent to the L1 contract. The contract interacts with the **ValueTransfer** contracts to move assets across chains.

4.2.3 Value Transfer contracts

The **ValueTransfer** contracts provide the actual pathway for moving tokens and ETH from L2 to L1. They act as adapters for the underlying bridging infrastructure, which vary for different chains. These include contracts to bridge ETH and wstETH.

- **EthArbValueTransfer and EthOPValueTransfer**: These contracts use the standard L2 bridges (specific to Arbitrum and Optimism) to transfer native ETH to the **xRenzoBridgeReceiver** contract on Ethereum mainnet.
- **LidoArbValueTransfer and LidoOPValueTransfer**: These contracts handle wstETH token bridging via the Lido protocol's gateway contracts on L2 networks.

4.2.4 Hyperlane messaging

A dedicated oracle contract (**RenzoOracleL2**) deployed on Arbitrum mainnet receives the price information from L1 using Chainlink. This price feed represents the exchange ratio between ezETH and ETH. The feed ensures that users receive an up-to-date minting rate on their deposits.

- **HyperlaneSender**: This contract queries the current exchange rate and the latest timestamp from the `RenzoOracleL2` contract and sends this data through Hyperlane to other L2 networks. The `HyperlaneReceiver` contract on each L2 receives the pricing info. The contract charges fees based on the gas estimates for cross-chain execution.
- **HyperlaneReceiver**: On each L2, the receiver processes incoming messages, validates the sender's legitimacy, and then calls the corresponding **xRenzoDepositNativeBridge** contract's **updatePrice(...)** function to be used for the next deposit operation. The synchronization mechanism ensures that price updates on L1 are reflected accurately on L2s, maintaining consistency when minting `ezETH` and `xezETH` regardless of the network.

4.3 Protocol flow

1. **Deposit on L2**: Users deposit native ETH or supported ERC20 tokens via the L2 deposit contracts, where a fee is deducted and xezETH is minted using the current exchange rate from the oracle.
2. **Asset transfer**: The `sweep(...)` function bridges the funds using the Value Transfer contracts, ensuring that the deposits are sent to L1.
3. **Processing assets on L1**: Once assets arrive on L1, the **xRenzoBridgeReceiver** processes deposits by unwrapping and converting assets as needed. Deposited funds are then forwarded to the Renzo **RestakeManager** contract, which mints `ezETH` tokens in return. The `ezETH` is sent to the protocol's Lockbox contract. The lockbox mints `xezETH` in return for the provided `ezETH` and sends the tokens to the `xRenzoBridgeReceiver` contract. The receiver immediately burns the `xezETH` since the tokens were already minted on L2.
4. **Price Synchronization**: The Hyperlane-based messaging system updates the price feed on L2s to match L1 conditions, ensuring that ongoing minting operations reflect correct asset valuations.
5. **Administration and monitoring**: Admins continuously monitor system components. Admins ensure that the funds are swept regularly and that the pricing information provided by the oracle is up to date.

5 Risk Rating Methodology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6 Issues

6.1 [Info] Incorrect configuration of token time discount on Arbitrum

File(s): contracts/Bridge/L2/xRenzoDepositNativeBridge.sol

Description: The `xRenzoDepositNativeBridge` contract implements a time-based discount mechanism through the `tokenTimeDiscountBasisPoints` mapping to account for the time when funds do not generate yield during the L2 to L1 bridging period.

When users deposit funds on L2, there's a time delay before these funds reach L1 and generate yield in the Renzo Protocol. To account for this period of missed yield generation, a discount is applied to the effective value of the deposited tokens (users receive less `ezETH` than they would if depositing directly on L1).

While this value is correctly configured for tokens on the Base L2 network (9 basis points for native ETH), it is incorrectly set to 0 on Arbitrum:

```
# @audit-info: Base L2 (correct configuration - 9 basis points)
$ cast call 0x766A4df76Cb091C41De22F75b6BD6733fb730357 \
  "tokenTimeDiscountBasisPoints(address)(uint)" \
  0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE \
  --rpc-url https://base.llamarpc.com

9

# @audit-info: Arbitrum (incorrect configuration - 0 basis points)
$ cast call 0x2EfA125B4fcD5B082a4233BA55b53FeB2Fb4Ff33 \
  "tokenTimeDiscountBasisPoints(address)(uint)" \
  0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE \
  --rpc-url https://1rpc.io/arb

0
```

With the discount set to `0` on Arbitrum, users depositing on this L2 do not have their tokens appropriately discounted for the bridging delay period. This results in Arbitrum users receiving more favorable exchange rates, leading to a slight economic advantage compared to users on other L2 chains.

Recommendation(s): Consider updating the time discount values on Arbitrum by calling the `updateTokenTimeDiscount(...)` function.

Status: Fixed

Update from the client: Configured with the correct value (9 bps) on arbitrum.

6.2 [Info] Incorrect handling of `WETH` in the `sweep(...)` function

File(s): contracts/Bridge/L2/xRenzoDepositNativeBridge.sol

Description: There's an inconsistency in how the `xRenzoDepositNativeBridge` contract handles WETH tokens between the `deposit(...)` and `sweep(...)` functions. In the `deposit(...)` function, WETH is unwrapped to native ETH before the deposit begins:

```
function deposit(
    IERC20 _token,
    uint256 _amountIn,
    // ...
) external nonReentrant returns (uint256) {
    // ...
    // @audit If the token is WETH, unwrap to ETH.
    if (_token == IERC20(address(weth))) {
            weth.withdraw(_amountIn);
    }
    // @audit ETH will be deposited.
    return _deposit(...);
}
```

The above code ensures that only native ETH is bridged to L1, and WETH tokens can only be bridged to L1 after unwrapping.

However, the `sweep(...)` function lacks similar handling for WETH. Even though wETH cannot be bridged to L1 directly, the value of the `valueTransferBridges` mapping for the WETH token is set to `EthArbValueTransfer` contract address. The same contract is also used for bridging native ETH deposits to L1 and is only designed to handle native ETH, not ERC20 tokens such as wETH. If a user donates WETH tokens to the `xRenzoDepositNativeBridge` and calls the `sweep(...)` function with the wETH address, the following will happen:

1. `xRenzoDepositNativeBridge` approves WETH to the bridge, but the `transferRemote(...)` function of the `EthArbValueTransfer` contract doesn't use this approval;
2. WETH tokens will remain in the `xRenzoDepositNativeBridge` contract;
3. The function will emit a `BridgeSwept` event with misleading information, suggesting funds were bridged when they were not;
4. Multiple calls to this function will unnecessarily increase the allowance to the bridge contract;

Recommendation(s): Consider setting WETH address in the `valueTransferBridges` mapping to the zero address since WETH shouldn't be bridged.

Status: Fixed

6.3 [Info] Incorrect oracle configuration for `wstETH` token on Base

File(s): `contracts/Bridge/L2/xRenzoDepositNativeBridge.sol`

Description: In the `xRenzoDepositNativeBridge` contract, price oracles determine the ETH value of deposited collateral tokens. For `wstETH` (wrapped staked ETH), the oracle should return the `wstETH/stETH` conversion rate, as `wstETH` will be unwrapped to stETH when bridged to L1 before being deposited into the Renzo protocol.

This conversion is deterministic and handled by smart contract logic rather than being subject to market fluctuations. On Arbitrum, the oracle is correctly configured to use this conversion rate. However, on Base, the oracle is incorrectly configured to return the `wstETH/ETH` market rate:

```
# Get wstETH oracle address from `xRenzoDepositNativeBridge` on Base:
$ cast call 0x766A4df76Cb091C41De22F75b6BD6733fb730357 \
  "tokenOracleLookup(address)(address)" \
  0xc1cba3fcea344f92d9239c08c0568f6f2f0ee452 \
  --rpc-url https://base.llamarpc.com

0x43a5C292A453A3bF3606fa856197f09D7B74251a

# Get Chainlink Oracle description
$ cast call 0x43a5C292A453A3bF3606fa856197f09D7B74251a \
  "description()(string)" \
  --rpc-url https://base.llamarpc.com

"WSTETH / ETH"

# Get Chainlink Oracle wstETH/ETH rate
$ cast call 0x43a5C292A453A3bF3606fa856197f09D7B74251a \
  "latestAnswer()(uint)" \
  --rpc-url https://base.llamarpc.com

1197860113406258300
```

By comparison, the Arbitrum `wstETH` price is `1199433727364042426` wei, which is slightly more advantageous.

The result is that users depositing `wstETH` on Base receive slightly less `xezETH` than users depositing the same amount of `wstETH` on Arbitrum.

Recommendation(s): Consider updating the price oracle for `wstETH` on Base to use the `wstETH/stETH` exchange rate.

Status: Fixed

Update from the client: wstETH oracle have been updated to use `wstETH/stETH` price from oracle contract at `0xB88BAc61a4Ca37C43a3725912B1f472c9A5bc061`.

6.4 [Info] The `LidoOPValueTransfer` contract unnecessarily grants the `wstETH` allowance to the `lidoBridge` address

File(s): `contracts/Bridge/L2/ValueTransfer/OP/LidoOPValueTransfer.sol`

Description: The `LidoOPValueTransfer` contract is used by the `xRenzoDepositNativeBridge` contract to bridge the `wstETH` tokens deposited by users on Base to the Ethereum mainnet. The `transferRemote(...)` function from the `LidoOPValueTransfer` contract calls the `withdrawTo(...)` function from the `L2ERC20TokenBridge` contract to bridge the `wstETH` tokens over the OP stack's Native Bridge.

```
function transferRemote(
    // ...
    address token,
    uint256 amount
) external payable returns (bytes32 transferId) {
    // ...
    if (token != address(wstETH)) revert InvalidTokenReceived();
    // @audit wstETH tokens wait to be bridged in the LidoOPValueTransfer contract.
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);

    // @audit-issue The wstETH allowance of the Lido bridge is unnecessarily
    // increased. The bridge contract does not utilize the allowance logic.
    IERC20(token).safeIncreaseAllowance(address(lidoBridge), amount);
    // ...

    // @audit The withdrawTo(...) function on the bridge calls the bridgeBurn(...)
    // function, which bypasses token allowances.
    lidoBridge.withdrawTo(token, l1xRenzoBridge, amount, 100000, extraData);
    // ...
}
```

The `withdrawTo(...)` function will burn the `wstETH` tokens from the `LidoOPValueTransfer` contract and issue the cross-chain message to the Ethereum mainnet. Since the `L2ERC20TokenBridge` contract has a special minter and burner permissions, it does not require any previous ERC20 allowance to burn the `wstETH` tokens. The allowance provided to this contract in the `transferRemote(...)` function will remain unutilized. It can be observed in the current Base deployment that the `LidoOPValueTransfer` contract currently has `1730e18 wstETH` allowance for the `L2ERC20TokenBridge` contract.

Recommendation(s): Consider adding additional test cases around sweeping the tokens on Base and after that removing the redundant call to the `increaseAllowance(...)` function.

Status: Fixed

Update from the client: Redundant call to the `increaseAllowance(...)` function removed at Code Ref.

6.5 [Best Practices] Incorrect comments

File(s): `contracts/Bridge/*`

Description: The codebase contains several comments which are incorrect. Below is a non-exhaustive list of such code comments:

- `EthArbValueTransfer.sol`: The NatSpec for the 2nd param of the `transferRemote(...)` function mentions that the token should always be `wstETH`, while the function handles native Ether only. This is also the case for `EthOPValueTransfer.sol`;
- `EthArbValueTransfer.sol`: In the `transferRemote(...)` function there is a comment `// Do not allow ETH`, while the function is supposed to exclusively handle native Ether. This is also the case for `EthOPValueTransfer.sol`;
- `xRenzoDepositNativeBridge.sol`: The NatSpec above the `setSupportedToken(...)` function doesn't describe the `_valueTransferBridge` param;
- `HyperlaneSender.sol`: The NatSpec comments above the `pause(...)` and `unpause(...)` functions are inverted;

Recommendation(s): To make the code more readable, consider removing or changing the incorrect comments outlined above.

Status: Fixed

Update from the client: Comments fixed at Code Ref.

6.6 [Best Practices] Unused events

File(s): contracts/Bridge/L2/xRenzoDepositNativeBridge.sol

Description: The codebase contains event declarations that are not utilized by the protocol contracts. The following is a non-exhaustive list of unused events:

- `xRenzoDepositNativeBridge.sol` contains unused `BridgeSweeperAddressUpdated` event;

Recommendation(s): Consider removing unused events to keep the code clean and readable.

Status: Fixed

Update from the client: Unused `BridgeSweeperAddressUpdated` event removed at Code Ref.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Renzo protocol documentation

The documentation for the Renzo protocol is comprehensive and well-structured, providing detailed information on its architecture and functionality. The official website includes thorough explanations of the protocol's features and use cases, allowing readers to understand the core mechanics of Renzo.

Additionally, the code itself is well-commented with extensive NatSpec annotations, making the codebase highly readable and facilitating a smooth review process. The Renzo team has made sure to provide clear, technical insights that helped the Nethermind Security team in understanding the project's inner workings and core logic.

The Renzo team was also proactive in making themselves available for sync calls to address any questions or concerns raised during the review. These discussions ensured that the security team had all the necessary context to perform a thorough analysis.

8 Test Suite Evaluation

The Renzo protocol's bridging smart contracts are undertested. The test suite does not catch changes to the business logic of the contracts. To assess the test coverage for these files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch to "kill" the mutant.

The following table outlines the results of the analysis performed on Renzo's bridging smart contracts. The first column lists the contracts that were tested using mutation testing. The second column indicates the number of mutants generated and how many were "slain" (i.e., caught by the test suite). The third column provides the percentage of mutants slain, reflecting the effectiveness of the test suite in covering the particular contract. The higher the score, the better the test suite is at finding bugs.

Contract	Mutants (slain / total generated)	Score
`LidoArbValueTransfer.sol`	1 / 10	10.00%
`xRenzoDepositNativeBridge.sol`	10 / 267	3.75%
`RenzoOracleL2.sol`	45 / 47	95.74%
`HyperlaneReceiver.sol`	14 / 16	87.50%
`xRenzoBridgeReceiver.sol`	30 / 34	88.24%
`HyperlaneSender.sol`	27 / 32	84.38%
`EthArbValueTransfer.sol`	1 / 7	14.29%
`EthOPValueTransfer.sol`	1 / 7	14.29%
`LidoOPValueTransfer.sol`	0 / 10	0.00%
Total	129 / 430	30.00%

Remarks about the Renzo Protocol Test Suite

The current test suite for the Renzo Protocol is minimal and does not provide adequate coverage for the deployed contracts. While some unit tests exist, they do not sufficiently verify the behavior of the system as a whole. In particular, there is a lack of end-to-end testing across core protocol flows.

While the transmission of pricing information from the oracle via Hyperlane is covered by tests, the core `xRenzoDepositNativeBridge` contract and associated Value Transfer contracts remain significantly undertested, leaving critical protocol flows without sufficient verification.

To improve the reliability of the protocol, it is recommended to begin developing end-to-end tests on a forked network. Given that most contracts are already deployed on mainnet, constructing realistic test scenarios should be feasible and would significantly increase confidence in the system's correctness.

Although the logic implemented in the contracts is relatively simple, expanding the test suite to include more cases—especially across boundaries between components—will help catch edge cases and regressions early.

8.1 Automated Tools

8.1.1 AuditAgent

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

9 About Nethermind

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

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

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

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

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

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

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

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

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

Learn more about us at nethermind.io.

General Advisory to Clients

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

Disclaimer

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