// Private Smart Contract Security Assessment 04.29.2024 - 05.22.2024

Protocol Withdrawals

Renzo Protocol

HALBORN

Protocol Withdrawals - Renzo Protocol

Prepared by: HALBORN

Last Updated 05/31/2024

Date of Engagement by: April 29th, 2024 - May 22nd, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	2	2	1	3	1

TABLE OF CONTENTS

1. Introduction

Renzo engaged our security analysis team to conduct a comprehensive security audit of their smart contract ecosystem. The primary aim was to meticulously assess the security architecture of Renzo's smart contracts to pinpoint vulnerabilities, evaluate existing security protocols, and offer actionable insights to bolster security and operational efficacy of their smart contract framework. Our assessment was strictly confined to the smart contracts provided, ensuring a focused and exhaustive analysis of their security features.

2. Assessment Summary

Our engagement with Renzo spanned a four-week period, during which we dedicated one full-time security engineer equipped with extensive experience in blockchain security, advanced penetration testing capabilities, and profound knowledge of various blockchain protocols. The objectives of this assessment were to:

- Verify the correct functionality of smart contract operations.

- Identify potential security vulnerabilities within the smart contracts.

The assessment unearthed several vulnerabilities and areas for enhancement:

1. **Reentrancy Guard Conflict (Critical)**: In the `OperatorDelegator` contract, interactions between `completeQueuedWithdrawal` and the `receive()` function are hindered by a reentrancy guard, which could block the intended flow of ETH back into the contract, disrupting the withdrawal process.

2. **Zero Deposit Amount (Critical)**: The `RestakeManager` contract may process a zero deposit amount if all tokens are allocated to fill a buffer, leading to transaction reversion and potential gas wastage, which deteriorates user experience.

3. **Inflation Vulnerability in TVL Calculation (Critical)**: The `calculateTVLs` function inaccurately includes direct transfers to `withdrawQueue`, potentially inflating TVL calculations and affecting the ezETH minting process, which could be exploited through sandwich attacks.

4. **Improper Strategy Verification (High)**: In the `OperatorDelegator` contract, the lack of strategy validation could result in incorrect strategy assignments, disrupting token strategy mappings and affecting total TVL calculations.

5. **Inadequate Validation of Token Price (High)**: The `xRenzoDeposit` contract fails to validate the `lastPrice` below 1 ETH, risking under-collateralization and leading to issuance of excessive xezETH.

6. **Inaccurate Reward Calculations (Medium)**: The `RestakeManager` does not correctly account for non-beacon chain ETH rewards in `getTotalRewardsEarned`, leading to inflated reward calculations.

7. **Unused Pause and Unpause (Low)**: The `WithdrawQueue` contract includes pause functionality that is not utilized in action validations, making the pause and unpause functions non-functional.

8. **Incorrect Share Queue Check (Low)**: The `OperatorDelegator` contract's `getTokenBalanceFromStrategy` uses an incorrect reference to queued shares, potentially leading to erroneous token balance reports.

9. **Inconsistent Price Fetching (Low)**: The `xRenzoDeposit` contract's `getRate` function may return outdated prices, whereas `getMintRate` uses fresher data, leading to inconsistencies that could impact financial decisions.

10. **Unreturned Excess Payment (Informational)**: The `xRenzoBridge` contract's `sendPrice` function does not refund excess Ether for overpayments, potentially leading to unintentional fund losses.

Addressing these issues will significantly enhance the security, functionality, and user trust in Renzo's smart contract ecosystem. Implementing the recommended changes will mitigate risks and align the operations with the best security practices.

3. Test Approach And Methodology

Our testing strategy employed a blend of manual and automated techniques to ensure a thorough evaluation. While manual testing was pivotal for uncovering logical and implementation flaws, automated testing offered broad code coverage and rapid identification of common vulnerabilities. The testing process included:

- A detailed examination of the smart contracts' architecture and intended functionality.

- Comprehensive manual code reviews and walkthroughs.

- Functional and connectivity analysis utilizing tools like Solgraph.

- Customized script-based manual testing and testnet deployment using Foundry.

This executive summary encapsulates the pivotal findings and recommendations from our security assessment of Renzo's smart contract ecosystem. By addressing the identified issues and implementing the recommended fixes, Renzo can significantly boost the security, reliability, and trustworthiness of its smart contract platform.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: Contracts

(b) Assessed Commit ID: e70e1be

(c) Items in scope:

- contracts/token/IEzEthToken.sol
- contracts/token/EzEthTokenStorage.sol
- contracts/token/EzEthToken.sol
- contracts/EigenLayer/libraries/BeaconChainProofs.sol
- contracts/EigenLayer/libraries/EIP1271SignatureUtils.sol
- contracts/EigenLayer/libraries/StructuredLinkedList.sol
- contracts/EigenLayer/libraries/Endian.sol
- contracts/EigenLayer/libraries/Merkle.sol
- contracts/EigenLayer/libraries/BytesLib.sol
- contracts/EigenLayer/interfaces/IDelegationManager.sol
- contracts/EigenLayer/interfaces/IPauserRegistry.sol
- contracts/EigenLayer/interfaces/ISignatureUtils.sol
- contracts/EigenLayer/interfaces/IEigenPodManager.sol
- contracts/EigenLayer/interfaces/IWhitelister.sol
- contracts/EigenLayer/interfaces/IEigenPod.sol
- contracts/EigenLayer/interfaces/IDelayedWithdrawalRouter.sol
- contracts/EigenLayer/interfaces/IETHPOSDeposit.sol
- contracts/EigenLayer/interfaces/IStrategyManager.sol
- contracts/EigenLayer/interfaces/IStrategy.sol
- contracts/EigenLayer/interfaces/IDelegationFaucet.sol
- contracts/EigenLayer/interfaces/IBeaconChainOracle.sol
- contracts/EigenLayer/interfaces/ISocketUpdater.sol
- contracts/EigenLayer/interfaces/IPausable.sol
- contracts/EigenLayer/interfaces/ISlasher.sol
- contracts/EigenLayer/interfaces/IAVSDirectory.sol
- contracts/RestakeManagerStorage.sol
- contracts/Bridge/L2/IxRenzoDeposit.sol
- contracts/Bridge/L2/xRenzoDeposit.sol
- contracts/Bridge/L2/Oracle/RenzoOracleL2Storage.sol
- contracts/Bridge/L2/Oracle/RenzoOracleL2.sol
- contracts/Bridge/L2/Oracle/IRenzoOracleL2.sol
- contracts/Bridge/L2/xRenzoDepositStorage.sol
- contracts/Bridge/L2/PriceFeed/CCIPReceiver.sol
- contracts/Bridge/L2/PriceFeed/ConnextReceiver.sol
- contracts/Bridge/xERC20/contracts/optimism/OptimismMintableXERC20Factory.sol
- contracts/Bridge/xERC20/contracts/optimism/OptimismMintableXERC20.sol

- contracts/Bridge/xERC20/contracts/XERC20Factory.sol
- contracts/Bridge/xERC20/contracts/XERC20Lockbox.sol
- contracts/Bridge/xERC20/contracts/XERC20.sol
- contracts/Bridge/xERC20/interfaces/IOptimismMintableERC20.sol
- contracts/Bridge/xERC20/interfaces/IXERC20Lockbox.sol
- contracts/Bridge/xERC20/interfaces/IXERC20Factory.sol
- contracts/Bridge/xERC20/interfaces/IXERC20.sol
- contracts/Bridge/L1/IxRenzoBridge.sol
- contracts/Bridge/L1/xRenzoBridge.sol
- contracts/Bridge/L1/xRenzoBridgeStorage.sol
- contracts/Bridge/Connext/core/IXReceiver.sol
- contracts/Bridge/Connext/core/IWeth.sol
- contracts/Bridge/Connext/core/IConnext.sol
- contracts/Bridge/Connext/integration/LockboxAdapterBlast.sol
- contracts/Bridge/Connext/libraries/TokenId.sol
- contracts/Bridge/Connext/libraries/LibConnextStorage.sol
- contracts/Oracle/Mantle/METHShim.sol
- contracts/Oracle/Mantle/METHShimStorage.sol
- contracts/Oracle/Mantle/IMethStaking.sol
- contracts/Oracle/RenzoOracle.sol
- contracts/Oracle/RenzoOracleStorage.sol
- contracts/Oracle/IRenzoOracle.sol
- contracts/Oracle/Binance/WBETHShim.sol
- contracts/Oracle/Binance/WBETHShimStorage.sol
- contracts/Oracle/Binance/IStakedTokenV2.sol
- contracts/Rewards/RewardHandlerStorage.sol
- contracts/Rewards/RewardHandler.sol
- contracts/RateProvider/IRateProvider.sol
- contracts/RateProvider/BalancerRateProviderStorage.sol
- contracts/RateProvider/BalancerRateProvider.sol
- contracts/Permissions/IRoleManager.sol
- contracts/Permissions/RoleManagerStorage.sol
- contracts/Permissions/RoleManager.sol
- contracts/Delegation/OperatorDelegator.sol
- contracts/Delegation/OperatorDelegatorStorage.sol
- contracts/Delegation/IOperatorDelegator.sol
- contracts/Deposits/DepositQueueStorage.sol
- contracts/Deposits/IDepositQueue.sol
- contracts/Deposits/DepositQueue.sol
- contracts/IRestakeManager.sol
- contracts/RestakeManager.sol
- contracts/Errors/Errors.sol
- contracts/Withdraw/IWithdrawQueue.sol
- contracts/Withdraw/WithdrawQueue.sol
- contracts/Withdraw/WithdrawQueueStorage.sol
- contracts/TimelockController.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- 5700b8d5700b8d
- 89a344b89a344b
- 9595
- 9696
- 9494
- 2e4f4c62e4f4c6
- 9797

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	2	1	3	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
REENTRANCY GUARD CONFLICT	Critical	SOLVED - 05/29/2024
ZERO DEPOSIT AMOUNT	Critical	SOLVED - 05/29/2024
IMPROPER STRATEGY VERIFICATION	High	SOLVED - 05/29/2024
INADEQUATE VALIDATION OF TOKEN PRICE	High	SOLVED - 05/29/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INACCURATE REWARD CALCULATIONS	Medium	SOLVED - 05/29/2024
UNUSED PAUSE AND UNPAUSE	Low	SOLVED - 05/29/2024
INCORRECT SHARE QUEUE CHECK	Low	SOLVED - 05/29/2024
INCONSISTENT PRICE FETCHING	Low	SOLVED - 05/29/2024
UNRETURNED EXCESS PAYMENT	Informational	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 REENTRANCY GUARD CONFLICT

// CRITICAL

Description

In the `OperatorDelegator` contract, the `completeQueuedWithdrawal` function involves withdrawing ETH using the strategy linked to `IS_NATIVE`. Upon completion of the withdrawal, the ETH is sent back to the contract address, which triggers the `receive()` function. Both the `completeQueuedWithdrawal` function and the `receive()` function are protected by the `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuardUpgradeable`.
This setup creates a potential issue: when `completeQueuedWithdrawal` results in a call to `receive()` due to the ETH transfer, the `nonReentrant` modifier on both functions will prevent the `receive()` function from executing if it's already in a non-reentrant state. This is because the `nonReentrant` modifier prevents multiple entries into functions marked by it from the same call chain, which would happen when ETH is sent to the contract upon withdrawal completion.
This reentrancy prevention leads to a revert of the transaction, disrupting the intended flow of ETH back into the contract and effectively blocking the withdrawal process.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:C (10.0)

Recommendation

To address this issue, it's essential to evaluate the necessity of the `nonReentrant` modifier on the `receive()` function. Given the controlled and trusted nature of the internal calls and interactions within your smart contracts, particularly between the `OperatorDelegator` and the delegation manager, consider the following adjustments:
1. Remove the `nonReentrant` Modifier from `receive()`:
- Since the `receive()` function is primarily designed to handle incoming ETH transfers that are a direct result of predefined and trusted contract operations (like withdrawals from the delegation manager), the risks of reentrancy attacks are minimal. Removing the `nonReentrant` modifier from `receive()` will prevent the undesired interaction blockage without significantly increasing the risk of reentrancy attacks, as the only expected, calls are from controlled contract operations.
By tailoring the use of the `nonReentrant` modifier to the specific needs and interaction patterns of your contract, you can maintain functionality while ensuring security and robustness in contract operations.

Remediation Plan

SOLVED: The **Renzo team** solved this issue.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/87/commits/5700b8dd6318b6ac1c7a50875a12b167da65f032

7.2 ZERO DEPOSIT AMOUNT

// CRITICAL

Description

The `deposit` function in the `RestakeManager` contract is designed to handle ERC20 token deposits into the protocol. However, there is a flaw where the `amount` of tokens intended to be deposited into an `OperatorDelegator` can inadvertently be set to zero due to all tokens being allocated to fill the withdraw buffer. This condition occurs when the deposited amount is equal to or less than the required amount to fill the buffer (`bufferToFill`).
The function checks the buffer deficit and adjusts the `amount` to be sent to the `OperatorDelegator` by reducing it by `bufferToFill`. If the original deposit `amount` equals `bufferToFill`, then the remaining `amount` becomes zero. The function then proceeds to `safeApprove` and attempts to execute `operatorDelegator.deposit(_collateralToken, amount);`, *which will fail if* `amount` is zero because the `OperatorDelegator`'s `deposit` function contains a guard against zero deposits (`InvalidZeroInput` error).
This issue can lead to failed transactions for users attempting to deposit amounts that do not exceed the current buffer deficit, resulting in an unnecessary consumption of gas and a poor user experience.

Proof of Concept

```
        address caller = msg.sender;

        // Give the caller some stETH.
        fs.stETH().deal(caller, 1000 ether);

        // Approve the restakeManager to spend the stETH.

        vm.startPrank(caller);
        fs.stETH().approve(address(fs.restakeManager()), 1000 ether);

        // NOTE: Buffer amount set to 10 ether
        fs.restakeManager().deposit(fs.stETH(), 5 ether);

        vm.stopPrank();
```

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:H/D:N/Y:N/R:N/S:U (9.4)

Recommendation

To resolve this issue and ensure robustness in deposit handling, consider implementing the following changes:

1. **Pre-Deposit Check**: Before adjusting the `amount` for the buffer fill, add a check to ensure that the deposit amount exceeds the buffer requirement. If the entire deposit amount is needed for the buffer, the function should either skip the call to the `OperatorDelegator`'s `deposit` function or provide a clear and specific error message indicating that the deposit amount is insufficient to proceed beyond filling the buffer.
2. **Conditional Execution**:
- Implement logic to only call `operatorDelegator.deposit(_collateralToken, amount);` *if* `amount` is greater than zero after adjusting for the buffer fill. This prevents attempting to deposit zero tokens, which is invalid.
- Use a conditional statement to skip `safeApprove` and the deposit call if `_amount` is zero.

```
if(_amount != 0) {
    // Approve the tokens to the operator delegator
    _collateralToken.safeApprove(address(operatorDelegator), _amount);

    // Call deposit on the operator delegator
    operatorDelegator.deposit(_collateralToken, _amount);
}
```

3. **Enhanced User Feedback**: Provide feedback to the user when a deposit is entirely used to fill the buffer, possibly through an event or revert message, to clarify why no tokens were deposited into the `OperatorDelegator`.
These modifications will help prevent failed transactions due to zero deposits, improve gas efficiency by avoiding unnecessary calls, and enhance the clarity of transaction outcomes for users.

Remediation Plan

SOLVED: The **Renzo team** solved this issue.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/87/commits/89a344ba837d98018b63a450666a72b0827aa8cf

7.3 IMPROPER STRATEGY VERIFICATION

// HIGH

Description

The `setTokenStrategy` function in the `OperatorDelegator` contract is responsible for assigning a strategy to a token. This function lacks proper validations, leading to potential misconfigurations that could disrupt the token strategy mapping and total TVL calculations:

1. **Strategy Validation**: The function does not verify if the `_strategy` address is valid or matches the underlying token. Setting an incorrect strategy that doesn't correspond to the token could lead to errors when interacting with the strategy, such as when calling `userUnderlyingView` on an incorrect or zero address, resulting in failed transactions or unexpected behaviors.
2. **Duplicate Strategy Assignments**: The function allows the same strategy to be used for different tokens. This could lead to duplicate counts in total TVL calculations if the same strategy manages balances for multiple tokens.

Proof of Concept

```
pragma solidity ^0.8.0;

import "ds-test/test.sol";
import "forge-std/Test.sol";

import "../fuzzing/FuzzSetup.sol";
import "@openzeppelin/contracts/proxy/Clones.sol";

contract StrategyMock {

    mapping(address => uint256) public userUnderlyingView;

    function mock_setUnderlayingView(address who, uint256 amount) public
{
        userUnderlyingView[who] = amount;
    }
}


contract TestOperatorDelegator is FuzzSetup {

    using Clones for address;

    function test_operator_delegator() public {

        OperatorDelegator od = OperatorDelegator(payable(address(new
```

```
OperatorDelegator()).clone()));
        od.initialize(
            roleManager,
            strategyManager,
            restakeManager,
            delegationManager,
            eigenPodManager
        );

        hevm.prank(ADMIN);
        restakeManager.addOperatorDelegator(
            od,
            1000
        );

        StrategyMock stStrategyMock = new StrategyMock();
        stStrategyMock.mock_setUnderlayingView(address(od), 1000);

        hevm.prank(ADMIN);
        od.setTokenStrategy(
            stETH,
            IStrategy(address(stStrategyMock))
        );

        hevm.prank(ADMIN);
        od.setTokenStrategy(
            rETH,
            IStrategy(address(stStrategyMock))
        );

        uint256 tokenBalanceFromStrategy_stETH =
od.getTokenBalanceFromStrategy(stETH);
        uint256 tokenBalanceFromStrategy_rETH =
od.getTokenBalanceFromStrategy(rETH);

        console.log("tokenBalanceFromStrategy_stETH: %s",
tokenBalanceFromStrategy_stETH);
        console.log("tokenBalanceFromStrategy_rETH: %s",
tokenBalanceFromStrategy_rETH);
    }
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation

To enhance the security and reliability of the `setTokenStrategy` function, consider implementing the following checks and controls:
- **Strategy Existence and Compatibility Check**: Before setting a strategy, verify that the strategy address is not zero and corresponds to the token. This can be done by querying the strategy for its expected token type and comparing it with the provided token.
These improvements will help prevent misconfigurations and ensure that the contract operates securely and as expected, mitigating risks associated with incorrect strategy assignments and calculations.

Remediation Plan

SOLVED: The **Renzo team** solved this issue. Duplication is prevented by the fact that the strategy token is verified, there shouldn't be more than 1 strategy with the same token otherwise EigenLayer will be broken.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/95

7.4 INADEQUATE VALIDATION OF TOKEN PRICE

// HIGH

Description

The `xRenzoDeposit` contract employs a dual-source price fetching mechanism via the `getMintRate` function, which can source data from either a `RenzoOracleL2` oracle or an internally stored `lastPrice`. The `RenzoOracleL2` ensures the price never falls below 1 ETH by design to prevent issues such as under-collateralization, which could lead to more xezETH being minted than what is represented by underlying assets on L1.
However, there is a significant oversight in the `xRenzoDeposit`'s handling of `lastPrice`. If the oracle is not present or the price from the `RenzoOracleL2` is not the latest, the contract relies on `lastPrice`, which does not undergo the same rigorous checks to ensure it does not fall below 1 ETH. This creates a potential vulnerability where an outdated or incorrectly set `lastPrice` could lead to the issuance of an excessive amount of xezETH relative to the collateral swept to L1, potentially causing financial discrepancies and impacting the system's integrity.

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation

To mitigate this vulnerability and ensure the robustness of financial operations across the L1 and L2 platforms, the following changes are recommended:
1. **Implement Minimum Price Validation in `_updatePrice`:**
- Adjust the `_updateType` function to include a validation check that ensures the new price is not below 1 ETH. This measure would prevent the introduction of potentially harmful price updates that could lead to under-collateralization issues.
2. **Centralize Price Validation Logic:**
- Extract the price validation logic into a separate internal function that both `_updatePrice` and any other method that modifies price-related state can call. This ensures consistency in how prices are validated across the contract.
3. **Enhance `sendPrice` Safeguards:**
- Ensure that the `sendPrice` function on `xRenzoBridge`, or any external interaction that might influence the price, includes checks to prevent the broadcast or acceptance of prices below 1 ETH. This will further protect against external manipulation or errors.
Implementing these recommendations will help maintain the integrity and stability of the `xRenzoDeposit` system by ensuring that all price data, regardless of its source, meets the necessary thresholds to protect against financial risks and maintain systemic health.

Remediation Plan

SOLVED: The **Renzo team** solved this issue.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/96

7.5 INACCURATE REWARD CALCULATIONS

// MEDIUM

Description

In the `RestakeManager` contract, the `getTotalRewardsEarned` function calculates the total amount of rewards earned by the protocol. The current implementation aggregates rewards from multiple sources, including native ETH and ERC20 tokens. However, it does not accurately account for non-beacon chain ETH rewards that are sent directly to the `EigenPod`. These funds, referred to as `nonBeaconChainETHBalanceWei`, are not derived from staking but may still be included in the total rewards calculation as part of the `EigenPod` balance.

This oversight leads to an inflated total rewards calculation because it includes funds that should not be classified as staking rewards. Although the function correctly aggregates balances from various sources, including the direct balances of tokens and ETH within the contract and those managed by operator delegators, it fails to subtract the non-beacon chain ETH balances which are not the result of staking activity but could include funds sent via contract `selfdestruct` or other non-standard transfers that do not trigger the `receive()` function.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation

To improve the accuracy of the rewards calculation in the `getTotalRewardsEarned` function, consider the following enhancements:

1. Subtract Non-Beacon Chain ETH Balances:
- Modify the function to subtract the `nonBeaconChainETHBalanceWei` from the balance reported by each `EigenPod` associated with operator delegators. This adjustment will ensure that only ETH balances resulting from beacon chain staking rewards are included in the total rewards calculation.
2. Rely on `EigenPodManager` Records for Precision:
- For complete accuracy, consider using records from the `eigenPodManager` to obtain a precise count of beacon chain rewards, excluding any funds added to the `EigenPods` through non-standard methods such as `selfdestruct`. This approach requires additional integration but offers the highest level of precision.
3. Enhance Data Integrity Checks:
- Implement checks to ensure that only beacon chain-derived rewards are included in the calculation. This could involve tracking the sources of deposits into the `EigenPods` and excluding any amounts that cannot be explicitly linked to beacon chain rewards.

These recommendations will help ensure that the `getTotalRewardsEarned` function provides a more accurate and reliable measure of the rewards generated by staking activities, enhancing financial reporting and decision-making processes within the protocol.

Remediation Plan

SOLVED: The **Renzo team** solved this issue by deprecating the functionality.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/94

7.6 UNUSED PAUSE AND UNPAUSE

// LOW

Description

In the `WithdrawQueue` contract, the functions `pause()` and `unpause()` are present and are correctly restricted to an administrator role through the `onlyWithdrawQueueAdmin` modifier. However, no functionality in the contract actively checks the paused state before executing actions. This oversight means that despite having the ability to toggle the pause state, the contract does not utilize this state to prevent or allow actions based on whether it is paused or active. Therefore, the `pause()` and `unpause()` functions are effectively non-functional and do not impact the contract's behavior as one might expect.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:U (3.1)

Recommendation

To address this issue, it is recommended to integrate the `paused()` state check in all sensitive functions that should be restricted when the contract is paused. This typically includes functions that handle asset transfers, state changes, or any interactions that should be temporarily halted for security or administrative reasons. For instance, adding the `whenNotPaused` modifier (from OpenZeppelin's `Pausable` contract) to critical functions such as `fillEthWithdrawBuffer`, `fillERC20WithdrawBuffer`, and `withdraw` can effectively enforce the paused state. This approach ensures that the pause functionality is meaningful and aligns with the security practices intended to protect the contract's integrity during critical operations or emergencies.

Remediation Plan

SOLVED: The **Renzo team** solved this issue. Claim cannot be called when the contract is paused.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/87/commits/2e4f4c6701ed004c2f10addde2d3089ac1b3d032

7.7 INCORRECT SHARE QUEUE CHECK

// LOW

Description

The function `getTokenBalanceFromStrategy` in the `OperatorDelegator` contract contains a logical error where it incorrectly checks the queued shares. The function is designed to retrieve the token balance from a specific strategy, factoring in any shares that are queued for withdrawal. The current implementation checks `queuedShares[address(this)] == 0` to determine if there are any shares queued for the calling contract itself, rather than checking the shares associated with the specific token in question.
This incorrect check can lead to erroneous balance calculations if there are shares queued for the token but the contract itself has no shares queued. The result is that the function may report balances that fail to account for pending withdrawals, potentially misleading other contract functions or external callers about the available balance, impacting decisions based on this balance (e.g., withdrawals, deposits, rewards calculations).

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation

To correct this issue, modify the `getTokenBalanceFromStrategy` function to properly reference the queued shares for the specific token being queried. The line:

```
queuedShares[address(this)] == 0
```

should be replaced with:

```
queuedShares[address(token)] == 0
```

This change ensures that the function checks for queued shares specifically associated with the token in question, thus providing accurate and reliable balance information. This fix will prevent potential mismanagement of token balances and ensure that all operations considering token balances are based on accurate data, especially in the context of withdrawals and financial calculations within the contract.

Remediation Plan

SOLVED: The **Renzo team** solved this issue.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/87/commits/89a344ba837d98018b63a450666a72b0827aa8cf

7.8 INCONSISTENT PRICE FETCHING

// LOW

Description

The `getRate` function in the `xRenzoDeposit` contract returns the `lastPrice`, which represents the most recent price updated either by an owner or cross-chain update. However, this method does not perform any checks to ensure the price is current or pull the latest available data from an external oracle, as done in the `getMintRate` function. The `getMintRate` function checks for the most recent timestamp between the internal last price update and an external oracle, using the freshest data available.
This discrepancy in how prices are fetched can lead to situations where `getRate` might return stale or outdated prices, while other parts of the contract that use `getMintRate` have access to more accurate, up-to-date values. Such inconsistencies can affect operations dependent on the price data, leading to potential inaccuracies in financial calculations, affecting minting of tokens, and potentially exposing the system to manipulation or errors based on outdated price information.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation

To ensure consistency in price data across the contract and reduce the risk of decisions being made on outdated information, the following adjustments are recommended:
1. **Update `getRate` to use `getMintRate` Logic**:
- Modify `getRate` to replicate the logic in `getMintRate`, where it checks the freshness of the internal price against the external oracle and returns the most current value. This will ensure that any consumer of `getRate` always receives the most accurate price available to the system.
Implementing these recommendations will help maintain the integrity and accuracy of the price data used by the `xRenzoDeposit` contract, ensuring that all functionalities relying on this data are based on the most reliable and current information available.

Remediation Plan

SOLVED: The **Renzo team** solved this issue.

Remediation Hash

https://github.com/Renzo-Protocol/Contracts/pull/97

7.9 UNRETURNED EXCESS PAYMENT

// INFORMATIONAL

Description

The `sendPrice` function in the `xRenzoBridge` contract handles sending out price feeds to other networks via Connext and CCIP. The function is marked as `payable`, allowing it to accept Ether. However, there are several issues related to how the Ether and LINK token payments are managed:

1. **Excess Ether Not Returned**: For the Connext transactions, a relayer fee is specified (`connextDestinationParam[i].relayerFee`), but the function does not return any excess Ether sent above this fee. This can result in unintended loss of funds if the sender sends more Ether than necessary.
2. **Function Still Payable for CCIP**: Although payments for CCIP messages are made in LINK tokens, the function still accepts Ether, leading to potential confusion or errors. Excess Ether sent to this function, intended for CCIP operations, is not utilized or returned.
3. **Insufficient Fee Handling**: If the Ether sent with the transaction is less than the required relayer fee for a Connext message, the function will utilize the contract's balance to cover the difference. This behavior can deplete the contract's Ether reserves unintentionally if not monitored and managed correctly.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (1.5)

Recommendation

To resolve these issues and enhance the robustness of the payment handling in the `sendPrice` function, consider implementing the following changes:

1. **Refund Excess Ether**:
- After processing each Connext and CCIP message, calculate the excess Ether sent (if any) and refund it to the sender. This can be achieved by comparing `msg.value` with the actual fees used and sending back the difference.
2. **Restrict Payable Functionality for CCIP**:
- Since CCIP uses LINK tokens for fees, modify the function to reject any Ether sent when processing CCIP messages or remove the `payable` attribute if Ether is not needed at all for this function.
3. **Validate and Handle Ether Payments**:
- Before proceeding with any Connext transactions, ensure that the Ether sent (`msg.value`) is sufficient to cover the relayer fees specified. If not, either revert the transaction to prevent depletion of contract funds or explicitly handle this scenario by limiting the transactions to the funds available.

By addressing these points, the `xRenzoBridge` contract can more safely and effectively manage cross-chain price feed operations, ensuring financial accuracy and protecting against unintended fund losses.

Remediation Plan

ACKNOWLEDGED: The **Renzo team** acknowledged this finding.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.