Renzo

Restaking Smart Contract Review

Version: 2.1

June, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Renzo smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Renzo smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Renzo smart contracts.

Overview

Renzo is a liquid restaking protocol built on top of EigenLayer.

Users deposit ETH or approved collateral tokens into `RestakeManager` in exchange for Renzo's liquid restaking token ezETH. The deposited tokens get deposited into EigenLayer through `OperatorDelegator`s, which handle all interactions with EigenLayer including strategy and EigenPod deposits, delegation to EigenLayer operators, and withdrawals.

Security Assessment Summary

Scope

The scope of this time-boxed review was strictly limited to the following directories/files at commit bb1dc9e:

Retesting was performed on commit 7dabe54

- contracts/Bridge/Connext
- contracts/Bridge/L1
- contracts/Bridge/L2
- contracts/Delegation/
- contracts/Deposits/
- contracts/Oracle/
- contracts/Permissions/
- contracts/RateProvider/
- contracts/Rewards/
- contracts/token/
- contracts/RestakeManager.sol

Note: third party libraries and dependencies, such as OpenZeppelin, were excluded from the scope of this assessment.

Approach

The review was conducted on the files hosted on the Contracts repository at commit bb1dc9e.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Coverage Limitations

Due to a time-boxed nature of this review, all documented vulnerabilities reflect best effort within the alloted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 20 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 5 issues.
- Medium: 4 issues.
- Low: 5 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Renzo smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
RENZO-01	Re-entrancy Modifier Misuse Causes Lost ETH	Critical	Resolved
RENZO-02	Malicious Validator Frontrunning Attack	High	Closed
RENZO-03	Exchange Rate Manipulation From Operator-Initiated Undelegation	High	Resolved
RENZO-04	Lost Funds Due To Specifying Wrong ETH Address In `completeQueuedWithdrawal`	High	Resolved
RENZO-05	Stuck Queued Withdrawal Caused By Incorrect Access Control	High	Resolved
RENZO-06	Native ETH Restake Admin Can Drain Funds By Manipulating `tx.gasprice`	High	Resolved
RENZO-07	Potentially Outdated ezETH/ETH Rate On L2 Can Cause Insolvency	Medium	Resolved
RENZO-08	Lower ezETH Mint Rate Due To Rounding	Medium	Resolved
RENZO-09	Incorrect `stakedButNotVerifiedEth` Accounting Can Overinflate TVL	Medium	Closed
RENZO-10	Optimistic xezETH Minting Can Lead To Undercollateralization	Low	Resolved
RENZO-11	Missing Validation In `setMaxDepositTVL` Function	Low	Resolved
RENZO-12	`_refundGas()` Can Prevent Sending ETH Rewards To OperatorDelegators	Low	Resolved
RENZO-13	Centralisation Risks	Informational	Resolved
RENZO-14	`OptimismMintableXERC20Factory` Can Set Incorrect Optimism Bridge	Informational	Closed
RENZO-15	Use Of Deprecated `safeApprove()` Function	Informational	Resolved
RENZO-16	Miscellaneous General Comments	Informational	Resolved
RENZO-17	Double Use Of Validator Public Key Can Re-introduce Security Issues	Medium	Closed
RENZO-18	Price Updates DOS After Price Staleness	Low	Closed
RENZO-19	Remove Collateral DOS By Transfer Of Dust Amount	Low	Closed
RENZO-20	TVL Calculation Inaccuracies	Informational	Closed

RENZO-01	Re-entrancy Modifier Misuse Causes Lost ETH		
Asset	`OperatorDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Due to the reuse of the `nonReentrant` modifier, ETH cannot be withdrawn from EigenLayer by the `OperatorDelegator` contract, causing any withdrawn ETH to be lost.

Both the `receive()` and `completeQueuedWithdrawal()` functions in `OperatorDelegator` have `nonReentrant` modifiers.

Since there is only a single `nonReentrant` guard in OpenZeppelin's `ReentrancyGuardUpgradeable` contract, functions with the `nonReentrant` modifier cannot call one another. Hence, it is impossible for `OperatorDelegator` to receive ETH in `completeQueuedWithdrawal()` function, which would be the case if any natively restaked ETH was withdrawn from the EigenLayer beacon chain ETH strategy.

Recommendations

Remove the `nonReentrant` modifier from the `receive()` function in `OperatorDelegator`.

Resolution

Recommended removal of the `nonReentrant` modifier from the `receive()` function was performed.

This issue has been addressed in commit 5700b8d.

RENZO-02	Malicious Validator Frontrunning Attack		
Asset	`DepositQueue.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A malicious validator operator whose validator is about to receive staked ETH from the `DepositQueue` via the beacon chain deposit contract can front-run the initial deposit to steal the staked ETH.

`DepositQueue::stakeETHFromQueue()` stakes 32 ETH into a validator on the beacon chain via the DepositContract. Renzo stakes with trusted third party operators who provide the necessary arguments for the `DepositContract::deposit()` function for each new validator:

```
function deposit(
    bytes calldata pubkey,
    bytes calldata withdrawal_credentials,
    bytes calldata signature,
    bytes32 deposit_data_root
) override external payable {
    ...
}
```

The `withdrawal_credentials` parameter sets the address that receives ETH whenever the validator's beacon chain withdrawals are processed. EigenLayer's `EigenPod` contract requires this value to be set as the `OperatorDelegator`'s assigned `EigenPod` address. This value can only be set once which occurs when the validator is first initialised and receives its first ETH deposit from the deposit contract.

Hence, it is possible for a malicious operator to front-run this call with a minimum of 1 ETH to provide their own `withdrawal_credentials` for the validator. The subsequent 32 ETH deposit from the `DepositQueue` will still be accepted as a top-up, but the different `withdrawal_credentials` will be ignored by the consensus layer.

Once the 32 ETH deposit has been applied, the malicious operator can withdraw the stake to their own withdrawal credentials address which they provided on their initial 1 ETH deposit.

The impact is rated as high as this allows a malicious node operator to steal 32 ETH from the deposit queue. The likelihood is rated as medium as node operators are partially trusted third parties who control the validator key.

Recommendations

Each validator's beacon chain withdrawal credentials should be verified before ETH is staked into them from the deposit queue.

A minimum of 1 ETH still needs to be deposited for the validator to be initialized, which should be provided by Renzo's third party operators. Renzo should not be providing the 1 ETH initial deposit as that is also vulnerable to the same front-running attack. Once the withdrawal credentials of that validator has been verified, Renzo can refund the 1 ETH back to the operators as well as stake the remaining 31 ETH to activate the validator.

Note that the 31 ETH will have to be staked directly via the deposit contract, as `EigenPod::stake()` only allows for 32 ETH deposits. Also keep in mind that the `OperatorDelegator`'s `stakedButNotVerifiedEth` accounting will also need to be updated to work correctly with the changed validator deposit flow in the recommendation above.

Resolution

The development team acknowledged the issue and provided the following comment:

> We currently use permissioned validators with commercial agreements - we will address this when the protocol upgrades in the future to allow a non-permissioned operator set.

RENZO-03	Exchange Rate Manipulation From Operator-Initiated Undelegation		
Asset	`OperatorDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The EigenLayer operator that all `OperatorDelegator`'s are delegated to can initiate queued withdrawals that forfeit all strategy tokens/ETH balances and hence, Renzo TVL staked in EigenLayer. This results in the lock up of funds and dramatically decreases the ezETH mint rate.

`DelegationManager::undelegate()` allows EigenLayer operators to undelegate a staker that is delegated to them. Doing so initiates a queued withdrawal on behalf of the staker for all shares in every strategy in EigenLayer.

Since this queued withdrawal is not executed through `OperatorDelegator::queueWithdrawals()`, the `queuedShares` mapping is not incremented and these withdrawals are not tracked, causing `OperatorDelegator::getTokenBalanceFromStrategy()` and `OperatorDelegator::getStakedETHBalance()` to return zero for every collateral token.

This results in a large amount of Renzo's TVL being excluded from the accounting, which can be used to manipulate the ezETH/ETH mint rate to be very small, such that the malicious EigenLayer operator can deposit a very small amount of ETH or collateral tokens to mint a large amount of ezETH. A flashloan can also be used to amplify the profits from the exploit.

Furthermore, since `queuedShares` are not incremented during undelegation, the queued withdrawals cannot be completed as decrementing `queuedShares` in `OperatorDelegator::completeQueuedWithdrawal()` will cause an underflow error in line [**282**] below:

```
278 for (uint256 i; i < tokens.length; ) {
        if (address(tokens[i]) == address(0)) revert InvalidZeroInput();
280
        // deduct queued shares for tracking TVL
282     queuedShares[address(tokens[i])] -= withdrawal.shares[i];
        ...
284 }
```

This makes it impossible to recover the queued withdrawn TVL.

Recommendations

To prevent ezETH/ETH mint rate manipulation from being exploited by externally queued withdrawals such as operator-initiated undelegations, the `OperatorDelegator` contract should keep track of `DelegationManager`'s `cumulativeWithdrawalsQueued` nonce for each `OperatorDelegator`. Both `getTokenBalanceFromStrategy()` and `getStakedETHBalance()` should revert if the `OperatorDelegator` and `DelegationManager` nonces are out of sync.

Keep in mind that these suggestions do not prevent the execution of the attack described above. However, they prevent the exploiter from profiting from the ezETH/ETH mint rate manipulation, as well as provide a way to recover from the

attack by allowing an admin to recover withdrawn funds.

Resolution

Recommended addition was made, except using the withdrawal root since this contains both the nonce and what it's derived from. This achieves the same result. As well as the addition of two checks in `calculateTVLs()` and `addOperatorDelegator()`.

This issue has been addressed in commit 210aba1.

An additional issue was detected during retesting, due to using a circuit breaker that reverts during `calculateTVLs()`. Since the `RestakeManager.depositTokenRewardsFromProtocol()` calls `calculateTVLs()` after undelegation, it is impossible to accrue protocol rewards or swept tokens.

RENZO-04	Lost Funds Due To Specifying Wrong ETH Address In `completeQueuedWithdrawal`		
Asset	`OperatorDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Specifying a token from another queued withdrawal while withdrawing from `beaconChainETHStrategy` results in accounting errors that prevents the completion of that queued withdrawal.

EigenLayer's `DelegationManager::completeQueuedWithdrawal()` function takes in an array of tokens that correspond to all the strategies that are being withdrawn from. These token addresses are checked to ensure that they match with the strategy in `StrategyBase::_beforeWithdrawal()`:

```
function _beforeWithdrawal(address recipient, IERC20 token, uint256 amountShares) internal virtual {
    require(token == underlyingToken, "StrategyBase.withdraw: Can only withdraw the strategy token");
}
```

These token addresses are also used in Renzo to decrement the `queuedShares` mapping for each corresponding token in `OperatorDelegator::completeQueuedWithdrawal()`:

```
for (uint256 i; i < tokens.length; ) {
    if (address(tokens[i]) == address(0)) revert InvalidZeroInput();

    // deduct queued shares for tracking TVL
    queuedShares[address(tokens[i])] -= withdrawal.shares[i];
    ...
}
```

However, EigenLayer ignores the provided token address when withdrawing from the `beaconChainETHStrategy`, allowing the ETH withdrawal to complete with any token address as input. This allows a native ETH restake admin through malicious intent or user error to decrement the queued shares of another queued withdrawal token instead, resulting in an underflow error that would prevent further token withdrawal of that token type from being completed, and hence causing the withdrawn funds to be irrecoverably lost.

Recommendations

Consider adding the following check to make sure that only the `IS_NATIVE` address can be used for `beaconChainETHStrategy` withdrawals:

```
if (address(tokens[i]) != IS_NATIVE) {
    if (withdrawal.strategies[i] == delegationManager.beaconChainETHStrategy()) {
        revert IncorrectStrategy();
    }
    ...
}
```

Resolution

The recommended additional check within `completeQueuedWithdrawal()` has been included.

This issue has been addressed in commit 22f71b3.

RENZO-05	Stuck Queued Withdrawal Caused By Incorrect Access Control		
Asset	`DepositQueue.sol & OperatorDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The call to `DepositQueue::fillERC20withdrawBuffer()` can only be called by `RestakeManager` and so will fail in `OperatorDelegator::completeQueuedWithdrawal()`, resulting in the withdrawal failing until the buffer is filled.

`OperatorDelegator::completeQueuedWithdrawal()` calls `DepositQueue::fillERC20withdrawBuffer()` if the `WithdrawQueue` has a positive buffer deficit for all withdrawn tokens:

```
// check if token is not Native ETH
if (address(tokens[i]) != IS_NATIVE) {
    // Check the withdraw buffer and fill if below buffer target
    uint256 bufferToFill = withdrawQueue.getBufferDeficit(address(tokens[i]));

    // get balance of this contract
    uint256 balanceOfToken = tokens[i].balanceOf(address(this));
    if (bufferToFill > 0) {
        bufferToFill = (balanceOfToken <= bufferToFill) ? balanceOfToken : bufferToFill;

        // update amount to send to the operator Delegator
        balanceOfToken -= bufferToFill;

        // safe Approve for depositQueue
        tokens[i].safeApprove(address(restakeManager.depositQueue()), bufferToFill);

        // fill Withdraw Buffer via depositQueue
        restakeManager.depositQueue().fillERC20withdrawBuffer(
            address(tokens[i]),
            bufferToFill
        );
    }
    ...
}
```

However, `DepositQueue::fillERC20withdrawBuffer()` has an `onlyRestakeManager` modifier on it and cannot be called by any operator delegators:

```
function fillERC20withdrawBuffer(
    address _asset,
    uint256 _amount
) external nonReentrant onlyRestakeManager {
    ...
}
```

Hence, withdrawals from EigenLayer for tokens that have a withdraw buffer deficit in `WithdrawQueue` cannot be completed.

Recommendations

Consider changing the access control for `DepositQueue::fillERC20withdrawBuffer()` to allow calls from operator delegators.

Resolution

As recommended, the removal of the `onlyRestakeManager` modifier from the `fillERC20WithdrawBuffer()` function was performed.

This issue has been addressed in commit 647bc4c.

RENZO-06	Native ETH Restake Admin Can Drain Funds By Manipulating `tx.gasprice`		
Asset	`DepositQueue.sol & OperatorDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The native ETH restake admin can drain ETH from `DepositQueue` and `OperatorDelegator` by manipulating the `tx.gasprice` to receive a higher gas refund. This can be profitably exploited by colluding with a block proposer.

The gas refunding mechanism in `DepositQueue` and `OperatorDelegator` uses `tx.gasprice` to calculate the amount of ETH to refund back to the caller:

DepositQueue

```
uint256 gasUsed = (initialGas - gasleft()) * tx.gasprice;
uint256 gasRefund = address(this).balance >= gasUsed ? gasUsed : address(this).balance;
```

OperatorDelegator

```
uint256 gasSpent = (initialGas - gasleft() + baseGasAmountSpent) * tx.gasprice;
adminGasSpentInWei[msg.sender] += gasSpent;
```

However, `tx.gasprice` returns the `effective_gas_price` which is calculated after EIP-1559 as follows:

```
priority_fee_per_gas = min(transaction.max_priority_fee_per_gas, transaction.max_fee_per_gas - block.base_fee_per_gas)
effective_gas_price = priority_fee_per_gas + block.base_fee_per_gas
```

Since `transaction.max_priority_fee_per_gas` and `transaction.max_fee_per_gas` are set by the transaction originator, it is possible to manipulate `tx.gasprice` by setting these values unrealistically high.

The transaction's priority fee is sent to the block proposer, so a malicious native ETH restake admin can collude with a block proposer such that the exploit can be profitable for both parties.

Recommendations

Consider using `block.basefee` instead of `tx.gasprice` to calculate the gas refund.

Resolution

The recommendation to substitute `tx.gasprice` with `block.basefee` was performed.

This issue has been addressed in commit c219682.

RENZO-07	Potentially Outdated ezETH/ETH Rate On L2 Can Cause Insolvency		
Asset	`contracts/Bridge/*`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Due to delays in ezETH/ETH price oracle updates and sweeping, xezETH on L2s will be undercollateralised by the amount of ezETH locked in the xezETH lockbox on L1. This can lead to insolvency if a large number of xezETH tokens are bridged back to L1.

There are two ways `xRenzoBridge` can receive the ezETH/ETH mint rate:

1. Via a Connext or Chainlink CCIP message from L1 via `xRenzoBridge::sendPrice()`
2. Via a custom Chainlink price feed on L2 that reads from `BalancerRateProvider::getRate()` on L1

Messages sent through Connext and Chainlink CCIP incur a delay as time is needed to finalise and verify messages. For Connext, the `ConnextReceiver` requires authenticated calldata as the `_originSender` is checked, hence the message goes through Connext's slow path which can take upwards of 4 hours to complete. Chainlink CCIP execution latency largely depends on Ethereum's block finality time, which is approximately 15 minutes.

A custom Chainlink price feed incurs less latency than sending messages via Connext and CCIP, but still unavoidably has minor discrepancies in the ezETH/ETH rate due to the set deviation threshold. The price feed will only update its price if the new price deviates by more than the threshold, which is currently set as 0.5% for Arbitrum's ezETH/ETH exchange rate feed. This value is too high for ezETH/ETH as the rate slowly appreciates over time as staking rewards are received. Hence, it is likely that the L2 price feed will lag behind the L1 rate.

Furthermore, there is another delay involved with sweeping nextWETH and initiating the Connext xCall. Sweepers are required to pay for Connext's relayer fees upfront, which is covered by depositors based on a flat fee of 0.05%. Relayer fees are variable and can be considerably higher than the collected bridge fees, hence sweepers may wait for a prolonged period of time for the accrued bridge fees to be enough to cover the relayer fee before sweeping the batched funds.

Due to these delays, the mint rate of xezETH on L2s will lag behind the true mint rate on L1. Since ezETH/ETH is monotonically increasing during normal behaviour, it's extremely likely that the mint rate on L2 at time of the L2 deposit is lower than on the mint rate on L1 at time of the L1 deposit. Users depositing on L2s will receive more xezETH than the amount of ezETH that will be minted and locked once the funds are sweeped and deposited into `RestakeManager` on L1. This results in an undercollateralisation of xezETH in the `xezETHLockbox`, which can lead to insolvency of the lockbox if the xezETH is bridged back to L1.

Recommendations

Similarly to RENZO-10, instead of optimistically minting xezETH based on a potentially outdated price, the use of xCall callbacks and a credit token can prevent insolvency by ensuring that the correct amount of xezETH is minted on L2s.

However, the use of a credit token negatively impacts user experience. Alternatively, to minimise the Chainlink price feed update delays, the deviation threshold can be reduced. As a reference, a deviation of 0.2% is used for Rocketpool's rETH/ETH exchange rate feeds.

Resolution

The recommendation to adjust the divergence threshold was implemented in commit a2a41d2.

RENZO-08	Lower ezETH Mint Rate Due To Rounding		
Asset	`RenzoOracle.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The ezETH mint amount calculation performs multiple divisions, resulting in rounding errors that cause `mintAmount` to be less than intended.

`RenzoOracle::calculateMintAmount()` calculates the ezETH `mintAmount` as follows:

```
// Calculate the percentage of value after the deposit
uint256 inflationPercentaage = (SCALE_FACTOR * _newValueAdded) /
    (_currentValueInProtocol + _newValueAdded);

// Calculate the new supply
uint256 newEzETHSupply = (_existingEzETHSupply * SCALE_FACTOR) /
    (SCALE_FACTOR - inflationPercentaage);

// Subtract the old supply from the new supply to get the amount to mint
uint256 mintAmount = newEzETHSupply - _existingEzETHSupply;
```

`inflationPercentaage` is calculated through a mulDiv, and then used again in another mulDiv to calculate `newEzETHSupply`. This results in more rounding errors that reduce the ezETH `mintAmount`.

Furthermore, due to less ezETH being minted, xezETH tokens on L2s end up being undercollateralised by ezETH tokens in the `xezETHLockbox` in L1.

Recommendations

Instead of performing two mulDivs and then a subtraction to calculate `mintAmount`, use the following expression to calculate `mintAmount` with one mulDiv:

```
uint256 mintAmount = (_existingEzETHSupply * _newValueAdded) / _currentValueInProtocol;
```

Resolution

The recommended change to the calculation for `mintAmount` was implemented.

This issue has been addressed in commit 6abc062.

RENZO-09	Incorrect `stakedButNotVerifiedEth` Accounting Can Overinflate TVL		
Asset	`DepositQueue.sol & OperatorDelegator.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `OperatorDelegator` contract maintains a counter for `stakedButNotVerifiedEth`. This counter tracks ETH staked on the beacon chain for validators that have not verified their withdrawal credentials on `EigenPod`. However, several situations arise where `stakedButNotVerifiedEth` does not correctly decrement back to 0, causing the `OperatorDelegator`'s staked ETH balance to be overinflated. This directly impacts TVL calculations used which are used to determine the ezETH/ETH mint rate.

The `OperatorDelegator::stakeEth()` function increments the `stakedButNotVerifiedEth` state variable to account for ETH that has been staked into a validator but has not had its withdrawal credentials verified on the `EigenPod`.

`stakedButNotVerifiedEth` is later decremented by the validator's current effective balance when the validator's withdrawal credentials are verified, since the staked ETH will be accounted for inside the `OperatorDelegator`'s `podOwnerShares` balance in `EigenPodManager`.

However, scenarios can arise where `stakedButNotVerifiedEth` is not correctly decremented back to 0 due to the validator's current effective balance being less than `stakedButNotVerifiedEth`. These scenarios are listed below:

1. Natively restaking more than 32 ETH into a single validator.
2. Verifying withdrawal credentials with a validator effective balance of less than 32 ETH either by:
   a. Submitting withdrawal credentials after the validator has exited.
   b. Any penalties that decrease validator effective balance below 32 ETH.

The first scenario would be achieved by calling `DepositQueue::stakeEthFromQueue()` with the same arguments more than once, increasing `stakedButNotVerifiedEth` above 32 ETH. A validator's effective balance is capped at 32 ETH, (`MAX_EFFECTIVE_BALANCE`), resulting in `stakedButNotVerifiedEth` not being decremented back to 0. Any ETH above `MAX_EFFECTIVE_BALANCE` will be partially withdrawn through `DelayedWithdrawalRouter` and sent back to the `DepositQueue` to be included in Renzo's TVL calculation again. This means that the extra ETH from subsequent staking attempts is double-counted in TVL.

The second scenario occurs if the validator has exited before its withdrawal credentials are verified in `EigenPod`. In this case, `validatorCurrentBalanceGwei = 0` and hence `stakedButNotVerifiedEth` is not decremented. Once the validator's full withdrawal has been processed via `OperatorDelegator::verifyAndProcessWithdrawals()`, the `OperatorDelegator`'s `podOwnerShares` will be credited with the 32 ETH amount that was initially staked, resulting in the 32 ETH being double-counted in TVL.

In a similar but less severe scenario, penalties such as inactivity can cause a validator's effective balance to drop below 32 ETH and `stakedButNotVerifiedEth` from being correctly decremented back to 0. Additionaly, it is worth mentioning that due to hysteresis, a validator's actual balance only needs to decrease below 31.75 ETH for the effective balance

to decrease to 31 ETH. Conversely, the validator's actual balance needs to increase above 32.25 ETH for the effective balance to recover back to 32 ETH. This lack of precision in validator effective balance also leads to inaccuracies in `stakedButNotVerifiedEth` accounting that cause misrepresentations in TVL.

Recommendations

To prevent the first scenario, consider adding checks to make sure that a staker does not get natively restaked into the same validator more than once.

One way to ensure this is to keep track of all `validatorPubkeys` that have been staked into in `DepositQueue`.

To prevent the second category of scenarios, consider keeping a mapping of each validator's `stakedButNotVerifiedEth` value instead of using one global variable. When `OperatorDelegator::verifyAndProcessWithdrawals()` is called, reset the verified validator's `stakedButNotVerifiedEth` value back to 0 instead of decrementing it by `validatorCurrentBalanceGwei`.

Resolution

After further discussions with the development team, a new recommendation of restricting `stakedButNotVerifiedEth` at a maximum of 32 ETH and checking whether the validator has had its withdrawal credentials verified in `EigenPod` was made and implemented.

This issue has been addressed in commit 45a3840.

However, during retesting, the testing team identified that overinflated TVL is still possible if the user staked, verified their withdrawal credentials, then staked again. This is due to the credentials have been verified and will not count towards their `stakedButNotVerifiedEth`. As such, this edge case still provides capability to exploit the aforementioned issue. The development team acknowledged this issue, and will aim to resolve this in a subsequent release.

RENZO-10	Optimistic xezETH Minting Can Lead To Undercollateralization		
Asset	`xRenzoDeposit.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Optimistically minting xezETH on L2 could result in scenarios where the xezETH is undercollateralised on L1.

`xRenzoDeposit::_deposit()` mints xezETH to the user before the ETH is bridged to L1 and deposited into Renzo through `xRenzoDeposit::xReceive()`. If the `xReceive()` call fails, then the ETH would not be deposited into Renzo and xezETH would be undercollateralised.

The `xReceive()` call can fail due to any of these reasons:

1. The maximum deposit TVL in `RestakeManager` has been reached
2. The amount of WETH received is 0
3. `xRenzoBridge` has reached its burn limit

Note that if Connext execution fails on the fast path, then it can be replayed and potentially succeed. However, failed execution on the slow path (after reconciliation) cannot be replayed. No funds are lost since the stuck xezETH can be recovered by calling `xRenzoBridge::recoverERC20()`

Recommendations

Instead of minting xezETH optimistically, consider executing an xCall callback using nested xCalls such that the tokens are only minted once `xRenzoDeposit` has confirmation that the deposit was successful.

Since this is a 3 step process (deposit on L2, sweep and deposit on L1, confirm on L2), a credit token can be minted to the user at time of deposit which can later be redeemed for xezETH once the confirmation has been received.

Resolution

The issue has been resolved by allowing a bridge admin to call `xReceive()` in the case of bridge failure.

The maximum deposit TVL restriction in `RestakeManager` has also been removed to reduce the likelihood of the deposit failing.

The resolution can be seen in PR #98.

RENZO-11	Missing Validation In `setMaxDepositTVL` Function		
Asset	`RestakeManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `setMaxDepositTVL()` function allows setting a new maximum deposit TVL (`maxDepositTVL`) without checking the current deposit queue size. This oversight can result in situations where the new `maxDepositTVL` is set lower than the current total deposits, allowing for staking operations to proceed despite exceeding the specified maximum TVL.

Recommendations

Consider implementing a check to compare the new `maxDepositTVL` with the current total deposits in the deposit queue.

Resolution

The issue has been resolved in PR #101.

RENZO-12	`_refundGas()` Can Prevent Sending ETH Rewards To OperatorDelegators		
Asset	`OperatorDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `_refundGas()` function can be exploited to perform a denial-of-service (DoS) attack on the ETH rewards distribution to `OperatorDelegators`.

This function calculates the gas refund based on the `tx.gasprice` and transfers the refund to the `NativeEthRestakeAdmin`. However, if the contract balance is insufficient to cover the gas refund, the function will revert, causing the entire transaction to fail.

A combination of several conditions are required to trigger this issue;

1. A force transfer must be created by a malicious or unaware user.
2. The `NativeEthRestakeAdmin` must trigger protocol reward accrual through the `receive()` function.
3. The `address(this).balance > remainingAmount`, where `remainingAmount` is the amount of protocol rewards.

With these conditions, all protocol rewards less than `address(this).balance` will revert.

Additionally, the functionality of `OperatorDelegator.receive()` being used for protocol rewards where the `tx.origin == NativeEthRestakeAdmin` is intended to catch partial withdrawals leaving EigenLayer through the `DelayedWithdrawalRouter`. The contract on EigenLayer provides several functions, some are intended for use only by `msg.sender` that has `_userWithdrawals[recipient]`. However, these functions are for convenience and provide no guarantees on access control. A non-privileged user is able to claim on behalf of the `NativeEthRestakeAdmin` by calling `claimDelayedWithdrawals(address, uint256)`. Doing so will fail the necessary gas refund conditional statement `adminGasSpentInWei[tx.origin] > 0`. Therefore, any user will always be able to frontrun the `NativeEthRestakeAdmin` in order to bypass gas refund to the admin.

Recommendations

Gas refunds should compare the refunded amount, with the protocol reward value. Following this, do an additional check to subtract gas refund from the `msg.value`, if the reward is larger than the gas refund. Otherwise, it should subtract the reward amount from the expected gas refund.

The below is an example, which can be used in the scenario that no TVL is expected to be locked in the `OperatorDelegator` for the purposes of gas refunds.

```
if (adminGasSpentInWei[tx.origin] > 0) {
    uint256 remainingAmount = msg.value >= adminGasSpentInWei[tx.origin]
                ? msg.value - adminGasSpentInWei[tx.origin]
                : 0;

    // If no funds left, return
    if (remainingAmount == 0) {
        adminGasSpentInWei[tx.origin] -= msg.value;
        return;
    }
}
```

The `OperatorDelegator` contract is not designed to hold ETH in the contract, the only way for the contract to have a non-zero TVL, currently, is through force transfer mechanisms. By providing a sweeper function, it is possible to clean out any additionally ETH, which is highly likely as a result of malicious activity.

Additionally, the development team should consider the possibility of front-running the gas refund. If undesirable, protections should be placed to avoid non-admin users calling protocol rewards. Alternatively, all rewards of a large enough value will trigger rewards.

Resolution

The issue has been resolved in PR #114 by setting the remaining amount to `address(this).balance`.

RENZO-13	Centralisation Risks
Asset	`/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The suite of Renzo contracts are subject to varying degrees of centralization. If systems operate as intended, and off-chain agreements are adhered to, the system's access control places some limits on malicious user capabilities. However, if trust assumptions are broken, and users that are trusted can act maliciously, many of the other discussed vulnerabilities in this report are increasingly consequential and more likely.

Some examples of centralization in Renzo contracts include;

1. `RestakeManager` : can add additional collateral tokens, add and remove operators and change TVL limits.
2. `NativeEthRestakeAdmin` : has a large set of highly sensitive functions, this includes: recovering tokens, queuing and completing withdrawals, verifying withdrawal credentials and refunding gas spent.
3. `OperatorDelegatorAdmin` : is able to set `OperatorDelegators` to specific operators on EigenLayer, this is a highly sensitive function, which could intentionally or accidentally be used to bypass certain off-chain agreements Renzo has with their EigenLayer Operators.
4. `Operators` : on EigenLayer are capable of force undelegating Renzo OperatorDelegators. Doing so has significant affects on the health of Renzo's TVL, assuming fixes are implemented for , operators could still trigger circuit breakers.

These trusted roles and relationships can be used maliciously, or more importantly in accidental ways to cause significant harm to the protocol.

Recommendations

The testing team recommends assessing the impact of trusted actors. Where possible minimise the potential impact, by reducing the set of actions that they may perform. Where offchain agreements exist, consider more verbose contractual protections. These could include explict guarantees that offchain validators can't forcefully `undelegate()` a Renzo `OperatorDelegator` .

Resolution

The entralisation risks are resolved by using a governance protocol. The development team have provided the following comment.

> For now `DEFAULT_ADMIN_ROLE` will be moved to Timelock controller and later on once DAO in place the access will be moved to DAO.

RENZO-14	`OptimismMintableXERC20Factory` Can Set Incorrect Optimism Bridge
Asset	`OptimismMintableXERC20Factory.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `OptimismMintableXERC20Factory` contract contains the `_deployOptimismMintableXERC20()` function, which is responsible for deploying an `OptimismMintableXERC20` contract. By convention, the first element of the `_bridges` array should be the Optimism bridge. However, this convention is not enforced within the contract, allowing for the possibility of setting an incorrect Optimism bridge.

Recommendations

Consider adding a check within the `_deployOptimismMintableXERC20()` function to ensure that the first element of the `_bridges` array is the correct Optimism bridge address.

Resolution

The development team have opted not to resolve this issue as there is no intention of using canonical bridges moving forward.

RENZO-15	Use Of Deprecated `safeApprove()` Function
Asset	`RestakeManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `RestakeManager` contract uses the `safeApprove()` from OpenZeppelin's `SafeERC20` library. However, `safeApprove()` is deprecated due to its susceptibility to unintended reverts, and OpenZeppelin recommends using `safeIncreaseAllowance()` and `safeDecreaseAllowance()` instead.

Recommendations

Consider refactoring the code to use `safeIncreaseAllowance()` and `safeDecreaseAllowance()` instead of `safeApprove()`.

Resolution

Uses of `safeApprove()` have been removed in PR #100.

RENZO-16	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Missing Length Check For TVLs And Operator Delegators**

   Related Asset(s): RestakeManager.sol

   The `chooseOperatorDelegatorForDeposit()` function assumes that the length of the `tvls` array matches the length of the `operatorDelegators` array. Although it is highly unlikely that these lengths would differ under normal conditions, it is a best practice to include a check to ensure they are equal.

   Consider adding a check at the beginning of the function to ensure that the length of the `tvls` array matches the length of the `operatorDelegators` array.

2. **Enhancement Of Events To Include Old And New Values**

   Related Asset(s): RestakeManager.sol

   The `RestakeManager` contract includes events for tracking changes to `OperatorDelegator` allocations. However, these events currently only emit the new values. Including both the old and new values in events that emit updates can improve the transparency and traceability of state changes within the contract.

   Consider modifying the events to include both the old and new values.

3. **Use Of Magic Numbers For Decimal Precision**

   Related Asset(s): RestakeManager.sol

   The `RestakeManager` contract uses a magic number to verify the decimal precision of collateral tokens. Specifically, it checks for a precision of 18 decimals directly hard-coded. It is best practice for code not to contain magic numbers.

   Define a constant for the required decimal precision and use this constant in the comparison.

4. **Premature Comments For Future Protocol Versions**

   Related Asset(s): EzEthTokenStorage.sol

   The `EzEthTokenStorage` contract contains comments that seem premature, discussing future versions of the protocol and the addition of new variables. While planning for future updates is good practice, having such comments in the code can be confusing and may lead to misinterpretation of the current state and usage of the contract.

   Consider removing comments that reference future versions of the protocol from the current implementation or clearly explain its purpose.

5. **Inconsistent Function Naming With Abbreviations**

   Related Asset(s): RestakeManager.sol

   The naming of functions within the `RestakeManager` contract shows inconsistency in using abbreviations. Specifically, some functions use the abbreviation "TVL" while others use "Tvl." This inconsistency can cause confusion and make the codebase harder to navigate and maintain.

Consider adopting a consistent naming convention for abbreviations throughout the codebase. Choose either "TVL" or "Tvl" and update all relevant function names to match this standard.

6. **Inconsistent And Missing NatSpec Comments**

   Related Asset(s): RestakeManager.sol, EzEthToken.sol

   The NatSpec comments within the `RestakeManager` and `EzEthToken` contracts are inconsistent in their terminology or missing entirely. Specifically, the following cases:

   - The comments for `RestakeManager.addCollateralToken()` and `removeCollateralToken()` refer to "restake manager," whereas other comments use "restake manager admin."
   - The `RestakeManager.setTokenTvlLimit()` function is missing NatSpec entirely.
   - The `EzEthToken._beforeTokenTransfer()` function has unfinished NatSpec.
   - The general contract NatSpec is inconsistent throughout the codebase with some having certain tags and explanations that others do not.
   - The `onlyTokenAdmin()` modifier in `EzEthToken` has a comment and function name inconsistency. If it only applies to pausing/unpausing, the function name should reflect this. If not, then the comment should reflect this.

   Ensure all NatSpec comments use consistent terminology, explanations and are not missing.

7. **Unsafe Coding Practices In Conditional Statements**

   Related Asset(s): RestakeManager.sol

   The code structure within various functions such as the `addOperatorDelegator()` employ conditional statements without curly braces, followed by the instruction to be executed on a separate line. This pattern introduces a risk of inadvertently adding additional statements that are meant to be conditional but end up executing unconditionally. Such coding practices are considered high risk due to the potential for introducing logic errors, without offering any significant benefits in terms of gas savings or efficiency.

   Consider refactoring the code to include curly braces for all conditional statements to clearly define the scope of the condition and its associated actions.

8. **Inconsistent Representation Of Native ETH**

   Related Asset(s): *.sol

   Throughout the codebase, inconsistent representations for native ETH are used. In some parts of the code, the zero address (`0x0`) is used to represent ETH, while in other parts, a constant `IS_NATIVE` is used.

   Standardize the representation of native ETH throughout the contract and across all related contracts. Use a consistent approach, such as defining a constant `IS_NATIVE` , and replace all occurrences of the zero address with this.

9. **Missing Event Emission For State Change**

   Related Asset(s): RestakeManager.sol

   The `setMaxDepositTVL()` function updates the critical state variable `maxDepositTVL` but does not emit an event to log this change.

   Consider adding an event emission to log the old and new values of `maxDepositTVL` whenever it is updated.

10. **Price Feed Sender Can Use Accidentally Sent Funds**

    Related Asset(s): xRenzoBridge.sol

    The `sendPrice()` function does not check `msg.value` or transfer LINK tokens from the price feed sender. Hence, any accidentally sent ETH or LINK tokens to the `xRenzoBridge` contract can be used to pay for the CCIP/Connext messages instead of being recovered.

    Consider adding a check for `msg.value` in the `sendPrice()` function. Also, consider adding a function that keeps track of any LINK funded to the `xRenzoBridge` contract, so that any accidentally sent LINK cannot be used to pay for the CCIP fee.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team have addresses the issues where deemed appropriate.

RENZO-17	Double Use Of Validator Public Key Can Re-introduce Security Issues		
Asset	/		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `OperatorDelegator` contract was updated since our original review to include a mapping of `validatorCurrentStakedButNotVerifiedEth`. This change was made to protect against various accounting discrepancies, see RENZO-09. However, reusing the same public key across different `OperatorDelegator` contracts will cause regression issues, introducing all the same accounting discrepancies.

The `stakeEth` tracks `validatorCurrentStakedButNotVerifiedEth` as a mapping of `validatorPubKeyHash` to non-verified eth. This mapping is not checked elsewhere in other contracts. Since the relationship between `RestakeManager` to `OperatorDelegator` is one to many, it is possible to have multiple `OperatorDelegator` contracts all containing their own `validatorCurrentStakedButNotVerifiedEth`. Since no reconciliation is done across the different OperatorDelegator contracts, it is possible to reuse the same public key across different instances of `OperatorDelegator` contracts.

Reuse of the same public key, will reintroduce the following issues;

1. RENZO-02
2. Double stake then verify overinflates TVL, see RENZO-09
3. An additional issue involving staking, verifying withdrawal credentials, then staking again inflating TVL. This was not identified in the original issue RENZO-09 but has the same impact. It is worth mentioning that this issue exists across individual instances of the `OperatorDelegator` contract as well.

Recommendations

The testing team recommends keeping a global registry of all public keys in use to insure the same key isn't used twice. Additionally, it is worth considering the solution provided in Resolution which would prevent the funds being trapped in withdrawal credentials that are potentially not accurate for that `OperatorDelegator`.

Resolution

The development team acknowledged the issue with the following comment;

> *We are going to accept the risk on this last issue as it would require our administrative functions (off-chain) to make a serious mistake.*

RENZO-18	Price Updates DOS After Price Staleness		
Asset	`xRenzoDeposit.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `xRenzoDeposit` contract relies on various checks and bounds to maintain healthy price updates from forced updates from owners, CCIP receivers and return data from external calls to oracle price feeds. If price feeds exceed a tolerable divergence price updates across the bridge will no longer be possible.

The following code snippet shows divergence checks done in all price updates on the `xRenzoDeposit` contract:

```
// Check for price divergence - more than 1 percent
if (
    (_price > lastPrice && (_price - lastPrice) > (lastPrice / 100)) ||
    (_price < lastPrice && (lastPrice - _price) > (lastPrice / 100))
) {
    revert InvalidOraclePrice();
}
```

Local contract price updates will only be used in the case that the external call to `oracle.getMintRate()` on line [**295**] returns a timestamp less than the current local price update.

Consider the event that a local price update occurs by 1 percent, followed by an update on the external oracle update of 1 percent (still within tolerable bounds). No subsequent local contract price update will be accepted since it is now roughly 2 percent larger than the original local contract price update.

Recommendations

After fetching prices from the external oracle price feed, ensure to update the local contract price to ensure that synchronization issues do not occur.

Resolution

Discussions with the development team note that:

> *[We] do not have any deployments using the Oracle price feed... this is the expected behavior*

RENZO-19	Remove Collateral DOS By Transfer Of Dust Amount		
Asset	`RestakeManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The contract `RestakeManager` has functionality for adding and removing collateral tokens. In changes for the original commit, additional functionality has been added that prevents collateral token removal. If a collateral token has balance present in the withdraw queue, it will revert during withdrawal.

This protection has been added to provide some integrity to internal accounting systems. However, anyone is able to transfer tokens directly to the Withdraw Queue. Transferring dust amounts will lead to inability to remove tokens.

This may lead to compromises in the event that the team need to remove tokens that are unsafe to use on Renzo.

Recommendations

Consider an approach where low value amounts of tokens or TVL can be ignored or transferred to an admin.

Resolution

The testing team acknowledged this issue, aiming to address this in a future release.

RENZO-20	TVL Calculation Inaccuracies
Asset	/
Status	**Closed:** See Resolution
Rating	Informational

Description

TVL calculations rely on funds being held either in `podOwnerShares` on EigenLayer, the `depositQueue` or the `withdrawQueue`. However, funds can also exist as a result of protocol rewards which are held in the `EigenPod` directly.

If an EigenPod earns significant funds, that TVL isn't captured by the TVL calculations until a withdrawal is processed and the funds are placed into the withdraw queue. This means that TVL could unexpectedly increase significantly after processing large rewards on a series of EigenPods.

Recommendations

In this situation, tracking EigenPod balances would lead to over inflated TVL during full withdrawals. The testing team recommends revisiting the different areas that TVL could be retained within Renzo and EigenLayer and how that TVL is deposited. Properly accounting for all TVL will require considering all the various ways funds can be deposited and withdrawn from the system.

Resolution

The testing team acknowledged this issue, aiming to address this in a future release.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `Foundry` framework was used to perform these tests and the output is given below.

```
Ran 1 test for test/RoleManager.t.sol:RoleManagerTest
[PASS] test_initialize() (gas: 1521718)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 88.23ms (831.75s CPU time)

Ran 2 tests for test/BalancerRateProvider.t.sol:BalancerRateProviderTest
[PASS] test_getRate() (gas: 385030)
[PASS] test_initialize() (gas: 1029580)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 89.56ms (1.50ms CPU time)

Ran 2 tests for test/METHShimTest.sol:METHShimTest
[PASS] test_initialize() (gas: 917325)
[PASS] test_variousFunctions() (gas: 954484)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 91.77ms (311.79s CPU time)

Ran 3 tests for test/RenzoOracleL2.t.sol:RenzoOracleL2Test
[PASS] test_getMintRate() (gas: 84432)
[PASS] test_initialize() (gas: 1365772)
[PASS] test_setOracleAddress() (gas: 56981)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 93.53ms (587.33s CPU time)

Ran 12 tests for test/WithdrawQueue.t.sol:WithdrawQueueTest
[PASS] test_claim() (gas: 724409)
[PASS] test_claim_reverts() (gas: 724445)
[PASS] test_fillERC20WithdrawBuffer() (gas: 308484)
[PASS] test_fillEthWithdrawBuffer() (gas: 74379)
[PASS] test_getAvailableToWithdraw() (gas: 2272242)
[PASS] test_getBufferDeficit() (gas: 1322828)
[PASS] test_initialize() (gas: 2805434)
[PASS] test_pauseAndUnpause() (gas: 801107)
[PASS] test_updateCoolDownPeriod() (gas: 91269)
[PASS] test_updateWithdrawBufferTarget() (gas: 189301)
[PASS] test_withdraw() (gas: 1317661)
[PASS] test_withdraw_reverts() (gas: 578498)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 98.98ms (12.16ms CPU time)

Ran 12 tests for test/DepositQueue.t.sol:DepositQueueTest
[PASS] test_depositETHFromProtocol() (gas: 965420)
[PASS] test_fillERC20WithdrawBuffer() (gas: 337280)
[PASS] test_initialize() (gas: 2208930)
[PASS] test_receive() (gas: 5964256)
[PASS] test_setFeeConfig() (gas: 2301322)
[PASS] test_setRestakeManager() (gas: 2279341)
[PASS] test_setWithdrawQueue() (gas: 2278920)
[PASS] test_stakeEthFromQueue() (gas: 16953798)
[PASS] test_stakeEthFromQueueMulti() (gas: 17075571)
[FAIL. Reason: Total TVL should not change after staking ETH twice: 9500000000000000000 != 6300000000000000000]
      ↪  test_stakeEthFromQueueMulti_sameODTwiceBeforeVerifyingCredentials() (gas: 586280)
[PASS] test_sweepERC20() (gas: 1333996)
[PASS] test_undelegateAndSweepERC20() (gas: 612456)
Suite result: FAILED. 11 passed; 1 failed; 0 skipped; finished in 101.40ms (10.90ms CPU time)

Ran 6 tests for test/RenzoOracle.t.sol:RenzoOracleTest
[PASS] test_fuzzCalculateRedeemAmount(uint256,uint256,uint256) (runs: 1003, : 1415305, ~: 1415438)
[PASS] test_initialize() (gas: 1411142)
[PASS] test_lookupTokenAmountFromValue() (gas: 1775930)
[PASS] test_lookupTokenValue() (gas: 1778649)
[PASS] test_lookupTokenValues() (gas: 1822877)
[PASS] test_setOracleAddress() (gas: 1746666)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 254.93ms (168.43ms CPU time)

Ran 2 tests for test/EzEthToken.t.sol:EzEthTokenTest
[PASS] test_initialize() (gas: 1572162)
```

```
[PASS] test_tokenTransfer() (gas: 1785023)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 70.84ms (1.25ms CPU time)

Ran 5 tests for test/XERC20.t.sol:XERC20Test
[PASS] test_burn() (gas: 1049504)
[PASS] test_initialize() (gas: 2867924)
[PASS] test_mint() (gas: 222250)
[PASS] test_setLimits() (gas: 216219)
[PASS] test_setLockBox() (gas: 38378)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 85.40ms (2.39ms CPU time)

Ran 13 tests for test/RestakeManager.t.sol:RestakeManagerTest
[PASS] test_addCollateralToken() (gas: 1795633)
[PASS] test_addOperatorDelegator() (gas: 5467279)
[PASS] test_calculateTVLs() (gas: 209)
[PASS] test_chooseOperatorDelegatorForDeposit() (gas: 1507338)
[PASS] test_chooseOperatorDelegatorForWithdraw() (gas: 1590320)
[PASS] test_deposit() (gas: 1396549)
[PASS] test_depositETH() (gas: 489325)
[PASS] test_getCollateralTokenIndex() (gas: 33007)
[PASS] test_removeCollateralToken() (gas: 1748871)
[PASS] test_removeCollateralToken_invalidTVL() (gas: 785719)
[PASS] test_removeOperatorDelegator() (gas: 4386578)
[PASS] test_setOperatorDelegatorAllocation() (gas: 4211655)
[PASS] test_setPaused() (gas: 73124)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 86.72ms (13.12ms CPU time)

Ran 7 tests for test/OptimismMintableXERC20.t.sol:OptimismMintableXERC20Test
[PASS] test_bridge() (gas: 19974)
[PASS] test_burn() (gas: 74679)
[PASS] test_burn_revertsIfInsufficientBalance() (gas: 35603)
[PASS] test_initialize() (gas: 2748630)
[PASS] test_mint() (gas: 81171)
[PASS] test_remoteToken() (gas: 19953)
[PASS] test_supportsInterface() (gas: 17889)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 73.22ms (470.75s CPU time)

Ran 4 tests for test/RewardHandler.t.sol:RewardHandlerTest
[PASS] test_forwardRewards(uint256) (runs: 1004, : 1191658, ~: 1191946)
[PASS] test_initialize() (gas: 1106446)
[PASS] test_receive(uint256) (runs: 1004, : 1167437, ~: 1168132)
[PASS] test_setRewardDestination(uint256) (runs: 1004, : 1255832, ~: 1256598)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 674.38ms (592.85ms CPU time)

Ran 4 tests for test/OptimismMintableXERC20FactoryTest.t.sol:OptimismMintableXERC20FactoryTest
[PASS] test_constructor() (gas: 1908970)
[PASS] test_deployOptimismMintableXERC20() (gas: 1105291)
[PASS] test_deployOptimismMintableXERC20_RevertsOnMismatchedLengths() (gas: 41477)
[PASS] test_deployOptimismMintableXERC20_RevertsOnNoBridges() (gas: 23892)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 73.97ms (2.15ms CPU time)

Ran 8 tests for test/XERC20Lockbox.t.sol:XERC20LockboxTest
[PASS] test_deposit() (gas: 301919)
[PASS] test_depositNative() (gas: 102861)
[PASS] test_depositNativeTo() (gas: 113115)
[PASS] test_depositTo() (gas: 303474)
[PASS] test_initialize() (gas: 646705)
[PASS] test_withdraw() (gas: 125015)
[PASS] test_withdrawTo() (gas: 123996)
[PASS] test_withdraw_Reverts() (gas: 201048)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 67.35ms (1.63ms CPU time)

Ran 3 tests for test/XERC20Factory.t.sol:XERC20FactoryTest
[PASS] test_deployLockbox() (gas: 1804175)
[PASS] test_deployXERC20() (gas: 8797746687697210778)
[PASS] test_initialize() (gas: 4732756)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 72.45ms (1.39ms CPU time)

Ran 9 tests for test/PoCs.t.sol:PoCs
[PASS] testFuzz_calculateMintAmount_roundingError_invariantsUpheld_Vuln(uint256,uint256,uint256) (runs: 1003, : 23979, ~: 23991)
```

```
[PASS] testFuzz_calculateMintAmount_roundingError_noAssumptions_Vuln(uint256,uint256,uint256) (runs: 1004, : 21910, ~: 21946)
[PASS] test_completeQueuedWithdrawal_ethReentrancy_Vuln() (gas: 2147601)
[PASS] test_completeQueuedWithdrawal_fillERC20WithdrawBufferAccessControl_Vuln() (gas: 1589853)
[PASS] test_completeQueuedWithdrawal_incorrectToken_Vuln() (gas: 3293635)
[PASS] test_refundGas_gasPriceManipulation() (gas: 530921)
[FAIL. Reason: Total TVL should not change after staking ETH into validator twice: 6300000000000000000 != 9500000000000000000]
      ↪  test_stakeEthFromQueue_artificialTVLInflation_Vuln() (gas: 2338733)
[FAIL. Reason: Total TVL should not change after withdrawal: 12700000000000000000 != 9500000000000000000]
      ↪  test_stakeEthFromQueue_wrongOD_Vuln() (gas: 3152584)
[PASS] test_verifyWithdrawalCredentials_exitValidatorBeforeVerifyCredentials_Vuln() (gas: 2102289)
Suite result: FAILED. 7 passed; 2 failed; 0 skipped; finished in 141.68ms (134.87ms CPU time)

Ran 13 tests for test/OperatorDelegator.t.sol:OperatorDelegatorTest
[PASS] test_activateRestaking() (gas: 5258955)
[PASS] test_completeWithdrawalTokensOnly() (gas: 6450952)
[PASS] test_deposit() (gas: 5677111)
[PASS] test_getStrategyIndex() (gas: 5943034)
[PASS] test_initialiseOperatorDelegator() (gas: 9306295)
[PASS] test_queueWithdrawalsNativeOnly() (gas: 8871043)
[PASS] test_queueWithdrawalsTokensOnly() (gas: 6196240)
[PASS] test_receiveAndRefundGas() (gas: 9994688)
[PASS] test_recoverTokens() (gas: 8988824)
[PASS] test_setBaseGasAmountSpent() (gas: 5303316)
[PASS] test_setDelegateAddress() (gas: 22810)
[PASS] test_setTokenStrategy() (gas: 5383549)
[PASS] test_withdrawNonBeaconChainETHBalanceWei() (gas: 8884161)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 572.78ms (10.49ms CPU time)

Ran 2 tests for test/LockboxAdapterBlast.fork.t.sol:LockboxAdapterBlastForkTest
[PASS] test_bridgeTo() (gas: 741628)
[PASS] test_constructor() (gas: 74011)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 45.06s (5.31s CPU time)

Ran 7 tests for test/ConnextReceiver.fork.t.sol:ConnextReceiverForkTest
[PASS] test_constructor() (gas: 794609)
[PASS] test_pause() (gas: 23300)
[PASS] test_setRenzoDeposit() (gas: 30841)
[PASS] test_unPause() (gas: 24662)
[PASS] test_updateCCIPETHChainSelector() (gas: 29220)
[PASS] test_updateXRenzoBridgeL1() (gas: 30940)
[PASS] test_xReceive() (gas: 1520973)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 49.43s (9.10s CPU time)

Ran 7 tests for test/CCIPReceiver.fork.t.sol:CCIPReceiverForkTest
[PASS] test_ccipReceive() (gas: 1513874)
[PASS] test_constructor() (gas: 884841)
[PASS] test_pause() (gas: 23167)
[PASS] test_setRenzoDeposit() (gas: 30944)
[PASS] test_unPause() (gas: 24640)
[PASS] test_updateCCIPETHChainSelector() (gas: 29399)
[PASS] test_updateXRenzoBridgeL1() (gas: 30974)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 49.44s (9.10s CPU time)

Ran 18 tests for test/xRenzoDeposit.fork.t.sol:xRenzoDepositForkTest
[PASS] testFuzz_getBridgeFeeShare(uint256,uint256) (runs: 1004, : 188275, ~: 188979)
[PASS] test_deposit() (gas: 676558)
[PASS] test_depositETH() (gas: 1120016)
[PASS] test_deposit_reverts() (gas: 1784762)
[PASS] test_getMintRate() (gas: 309422)
[PASS] test_getMintRate_reverts() (gas: 156667)
[PASS] test_getRate() (gas: 193782)
[PASS] test_initialize() (gas: 3310496)
[PASS] test_recoverERC20() (gas: 862328)
[PASS] test_recoverNative() (gas: 213862)
[PASS] test_setAllowedBridgeSweeper() (gas: 189101)
[PASS] test_setOraclePriceFeed() (gas: 184230)
[PASS] test_setReceiverPriceFeed() (gas: 184291)
[PASS] test_sweep() (gas: 725187)
[PASS] test_updateBridgeFeeShare() (gas: 184518)
[PASS] test_updatePrice() (gas: 201211)
```

```
[PASS] test_updatePriceByOwner() (gas: 182566)
[PASS] test_updateSweepBatchSize() (gas: 186992)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 51.64s (18.84s CPU time)

Ran 5 tests for test/xRenzoBridge.fork.t.sol:xRenzoBridgeForkTest
[PASS] test_initialize() (gas: 3954944)
[PASS] test_recoverERC20() (gas: 764232)
[PASS] test_recoverNative() (gas: 113231)
[PASS] test_sendPrice() (gas: 2447742)
[PASS] test_xReceive() (gas: 2630085)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 53.34s (25.16s CPU time)

Ran 2 tests for test/PoCs.fork.t.sol:PoCsFork
[PASS] test_calculateMintAmount_underCollateralization_Vuln() (gas: 2698747)
[PASS] test_deposit_optimisticMinting_Vuln() (gas: 3617228)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 55.57s (29.99s CPU time)

Ran 2 tests for test/WBETHShim.t.sol:WBETHShimTest
[PASS] test_initialize() (gas: 916914)
[PASS] test_variousFunctions() (gas: 954069)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 56.01s (452.92s CPU time)

Ran 24 test suites in 57.40s (363.24s CPU time): 146 tests passed, 3 failed, 0 skipped (149 total tests)

Failing tests:
Encountered 1 failing test in test/DepositQueue.t.sol:DepositQueueTest
[FAIL. Reason: Total TVL should not change after staking ETH twice: 9500000000000000000 != 6300000000000000000]
      ↪  test_stakeEthFromQueueMulti_sameODTwiceBeforeVerifyingCredentials() (gas: 586280)

Encountered 2 failing tests in test/PoCs.t.sol:PoCs
[FAIL. Reason: Total TVL should not change after staking ETH into validator twice: 6300000000000000000 != 9500000000000000000]
      ↪  test_stakeEthFromQueue_artificialTVLInflation_Vuln() (gas: 2338733)
[FAIL. Reason: Total TVL should not change after withdrawal: 12700000000000000000 != 9500000000000000000]
      ↪  test_stakeEthFromQueue_wrongOD_Vuln() (gas: 3152584)

Encountered a total of 3 failing tests, 146 tests succeeded
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

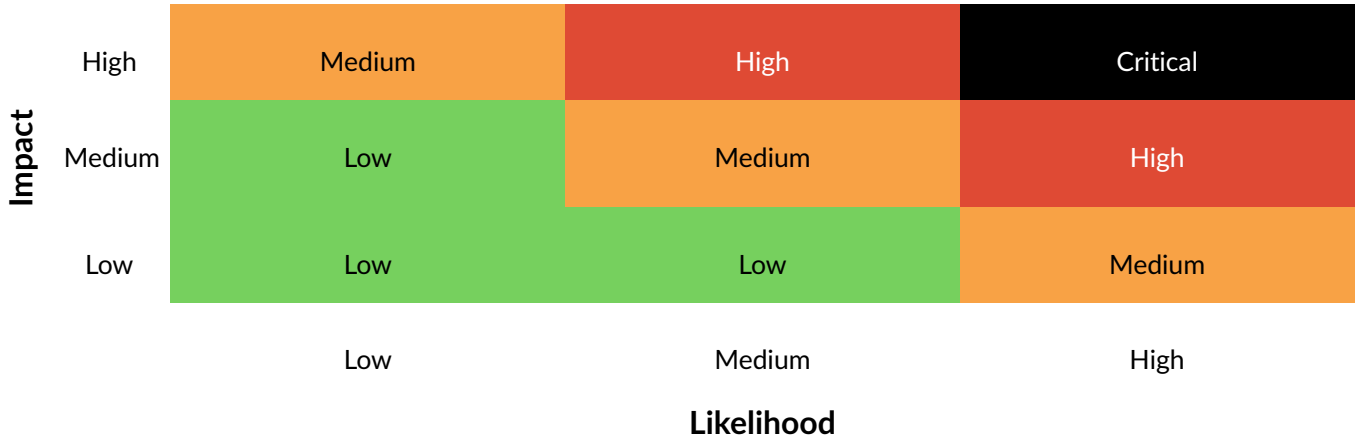

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].