Renzo Protocol - EVM Contracts

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **November 13th, 2023 - November 29th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	11/13/2023
0.2	Document Edits	11/28/2023
0.3	Draft Review	11/29/2023
1.0	Remediation Plan	11/30/2023
1.1	Remediation Plan Review	12/01/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The Renzo Protocol is a layer over top of EigenLayer (EL) that allows a community to pool their staked tokens together and earn rewards.

Renzo Protocol engaged Halborn to conduct a security assessment on their smart contracts beginning on November 13th, 2023 and ending on November 29th, 2023. The security assessment was scoped to the smart contracts provided in the Renzo-Protocol/Contracts GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided 17 days for the engagement and assigned a team of two full-time security engineers to review the security of the smart contracts in scope. The security team consists of a blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security risks, that were successfully addressed by Renzo Protocol. The main one was the following:
- Use the OpenZeppelin's SafeERC20 wrapper with the IERC20 interface in the sweepERC20() function of the DepositQueue contract.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (Slither).
- Testnet deployment (Foundry, Brownie).

2. 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.

2.1 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:

Exploitability 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
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:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

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

EXECUTIVE OVERVIEW

2.3 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.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	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

2.4 SCOPE

Code repositories:

1. Renzo Protocol

- Repository: Renzo-Protocol/Contracts
- Commit ID: 04bb57a5d32fa2da93e2c2068522c78bcf776913
- Smart contracts in scope:
    1. contracts/RestakeManager.sol
    2. contracts/RestakeManagerStorage.sol
    3. contracts/token/EzEthTokenStorage.sol
    4. contracts/token/EzEthToken.sol
    5. contracts/Delegation/OperatorDelegatorStorage.sol
    6. contracts/Delegation/OperatorDelegator.sol
    7. contracts/Oracle/RenzoOracle.sol
    8. contracts/Oracle/RenzoOracleStorage.sol
    9. contracts/Permissions/RoleManager.sol
    10. contracts/Permissions/RoleManagerStorage.sol
    11. contracts/Deposits/DepositQueueStorage.sol
    12. contracts/Deposits/DepositQueue.sol
- RewardHandler.sol 67baeb53c82ad7a665a430bac01c87a6dd95d0df

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	1	5	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) INCONSISTENT DECIMAL HANDLING IN ORACLE VALUE LOOKUP	Critical (10)	SOLVED - 11/30/2023
(HAL-02) MINTING LOGIC FLAW IN DEPOSIT FUNCTION FOR INITIAL SUPPLY	Critical (10)	SOLVED - 11/30/2023
(HAL-03) USING TRANSFER INSTEAD OF SAFETRANSFER	Medium (5.0)	SOLVED - 11/30/2023
(HAL-04) UNHANDLED EMPTY OPERATOR DELEGATORS LIST	Low (2.5)	SOLVED - 11/30/2023
(HAL-05) DEPOSIT/WITHDRAWAL RELIANCE ON CONTRACT BALANCE	Low (2.5)	SOLVED - 11/30/2023
(HAL-06) LACK OF FEE VALIDATION	Low (2.5)	SOLVED - 11/30/2023
(HAL-07) MISSING ZERO ADDRESS CHECKS	Low (2.5)	SOLVED - 11/30/2023
(HAL-08) TRANSFER ALLOWED DURING PAUSE FOR MINTER/BURNER	Low (2.5)	SOLVED - 11/30/2023
(HAL-09) SYNCHRONIZED OPERATOR DELEGATOR ADDITION AND ALLOCATION SETTING	Informational (1.5)	SOLVED - 11/30/2023
(HAL-10) MISSING DEPENDENCY INITIALIZATION	Informational (0.8)	SOLVED - 11/30/2023
(HAL-11) MISSING EVENTS FOR CONTRACT OPERATIONS	Informational (0.8)	SOLVED - 11/30/2023
(HAL-12) FOR LOOPS CAN BE GAS OPTIMIZED	Informational (0.0)	SOLVED - 11/30/2023
(HAL-13) USING REVERT STRINGS INSTEAD OF CUSTOM ERRORS	Informational (0.0)	SOLVED - 11/30/2023
(HAL-14) INCOMPLETE NATSPEC DOCUMENTATION	Informational (0.0)	SOLVED - 11/30/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) INCONSISTENT DECIMAL HANDLING IN ORACLE VALUE LOOKUP - CRITICAL(10)

Description:

In the lookupTokenValue and lookupTokenAmountFromValue functions of the RenzoOracle contract, there is an issue with the handling of decimal places when retrieving token values from an oracle. The function does not account for the varying decimal places used by different tokens and the oracle. As demonstrated in the provided proof of concept, when querying the value of two tokens with different decimal places (one with standard 18 decimals and another with 8), the function returns values that are not normalized to the same factor. This inconsistency leads to incorrect calculations, especially when aggregating values for total value locked (TVL) computations, where a uniform scale is crucial.

The issue arises because the oracle's price and the tokens' balances are used directly in the calculation without adjusting for their respective decimal places. This oversight leads to disproportionate values, as seen in the output of the test function, where one token's value is represented correctly in terms of ETH (with 18 decimals), but the other is not.

Proof of Concept:

Listing 1

```
function test_lookupTokenValue_invalidDecimals() external {
    vm.warp(1000000000);
    address admin = address(0x1337);
    vm.label(admin, "admin");

    MockRoleManager manager = new MockRoleManager();

    RenzoOracle oracle_implementation = new RenzoOracle();

    RenzoOracle oracle = RenzoOracle(address(new ERC1967Proxy(
```

```
address(oracle_implementation), abi.encodeWithSelector(
oracle_implementation.initialize.selector, address(manager)))));
        TestingOracle toracle = new TestingOracle();
        MyToken token = new MyToken();
        MyToken8Decimals token2 = new MyToken8Decimals();

        oracle.setOracleAddress(token, AggregatorV3Interface(address(toracle)));
        oracle.setOracleAddress(token2, AggregatorV3Interface(address(toracle)));

        // We assume in this case that the oracle is using 18 decimals, like many chainlink ETH feeds
        // For ETH feeds it is usually 18 decimals, other 8 decimals
        // In this case 1 "token" is 2000 ETH
        toracle.setDecimals(18);
        toracle.setLatestRoundData(1, 2000 * 10**18, 1000000000, 1000000000, 1);


        // All of those lookup calls should return the same value
        // The value should be represented in ETH 10**18 factor which means that
        // the value should return 2000 * 10**18: 2000000000000000000000

        // This one has 18 decimals, value returned is 2000000000000000000000
        console.log(oracle.lookupTokenValue(token, 1 * 10**token.decimals()));
        // This one has 8 decimals, value returned is 200000000000
        console.log(oracle.lookupTokenValue(token2, 1 * 10**token2.decimals()));

        assertEq(oracle.lookupTokenValue(token, 1 * 10**token.decimals()), oracle.lookupTokenValue(token2, 1 * 10**token2.decimals()));

    }
```

Output:

Listing 2

```
Logs:
  20000000000000000000000
  200000000000
  Error: a == b not satisfied [uint]
        Left: 20000000000000000000000
       Right: 200000000000
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:C/R:N/S:U (10)

Recommendation:

To effectively address the decimal inconsistency issue in the RenzoOracle contract's lookupTokenValue and lookupTokenAmountFromValue functions, it's recommended to incorporate a utility function, like scalePrice, for normalizing values (https://docs.chain.link/data-feeds/using-data-feeds). This function will adjust the oracle price and token balance to a common decimal scale, facilitating accurate calculations for TVL and related metrics. The approach involves:

1. **Implementing the scalePrice Function**: Integrate the provided scalePrice function into the contract. This function dynamically adjusts the price based on the difference in decimals between the oracle's price and the token's decimals. It scales up or down the price to ensure it matches the desired decimal scale.
2. **Adjusting Oracle Price**: In the lookupTokenValue function, apply scalePrice to the price returned by the oracle. Pass the oracle's decimal places and the desired scale (typically 18 decimals for Ethereum tokens) as arguments to scalePrice. With this change, you will be adding support for oracles that are not using standard 18 decimals for Ethereum tokens, if any. Other Chainlink oracles usually use 8 decimals.
3. **Adjusting Token Balance**: If the token has a different decimal

scale than the desired scale (e.g., 18 decimals), apply a similar adjustment to the token balance. This can be done within the lookupTokenValue function or through a separate utility function similar to scalePrice.

4. **Modified Calculation in lookupTokenValue**: With these adjustments, the oracle price and token balance will be on a consistent scale. The calculation within lookupTokenValue should then multiply the normalized price by the normalized balance and divide by the scale factor (typically 10**18 for Ethereum tokens).

Listing 3

```
function scalePrice(
    uint256 _price,
    uint8 _priceDecimals,
    uint8 _decimals
) internal pure returns (uint256) {
    if (_priceDecimals < _decimals) {
        return uint256(_price) * uint256(10 ** uint256(_decimals - _priceDecimals));
    } else if (_priceDecimals > _decimals) {
        return uint256(_price) / uint256(10 ** uint256(_priceDecimals - _decimals));
    }
    return _price;
}

function lookupTokenValue(IERC20 _token, uint256 _balance) public view returns (uint256) {
    ....
    return scalePrice(uint256(price) * _balance, IERC20Halborn(address(_token)).decimals() + oracle.decimals(), 18);
}
```

5. **Verify token and oracle decimals**: When adding both tokens and oracles it is recommended to check for decimals being a specific amount, 18 in this case. If this is performed, previous changes shouldn't be required and calculations on lookup values can be simplified.

By implementing the scalePrice function and these adjustments, the

RenzoOracle contract will return accurate and normalized values for different tokens, ensuring consistency in TVL calculations across various tokens with different decimal configurations. This approach enhances the reliability and accuracy of financial computations within the contract.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue by limiting both the token and oracle to 18 decimals when added, in commit 2c2e7e4006a2c5ef0623bc7fef30a77d7d49d059.

4.2 (HAL-02) MINTING LOGIC FLAW IN DEPOSIT FUNCTION FOR INITIAL SUPPLY - CRITICAL(10)

Description:

In the deposit function of the RestakeManager contract, there is an issue with the calculateMintAmount function from renzoOracle, particularly when ezETH.totalSupply() is zero. The current logic in calculateMintAmount is designed to mint tokens equal to _newValueAdded when _currentValueInProtocol is zero. However, it does not correctly handle the scenario where _currentValueInProtocol is greater than zero and ezETH.totalSupply() is zero.

In such a case, the function will not mint any tokens, even though new value is being added to the protocol. This situation can occur if the operatorDelegator already holds funds or specific tokens, leading to a non-zero _currentValueInProtocol. As a result, the initial depositors might not receive any ezETH tokens, despite contributing collateral to the protocol.

Proof of Concept:

Listing 4

```
contract MockRenzoOracle {
    uint256 constant SCALE_FACTOR = 10 ** 18;

    function lookupTokenValue(IERC20 _token, uint256 _balance) external view returns (uint256) {
        return 2000 * 10**18;
    }

    function calculateMintAmount(uint256 _currentValueInProtocol, uint256 _newValueAdded, uint256 _existingEzETHSupply) external pure returns (uint256) {
        // For first mint, just return the new value added
```

```
        if (_currentValueInProtocol == 0) {
            return _newValueAdded; // value is priced in base units, so divide by scale factor
        }

        // 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
        return newEzETHSupply - _existingEzETHSupply;
    }

}

contract MockOperatorDelegator {

    function deposit(address _token) external returns (uint256) {
    }

    function getTokenBalanceFromStrategy(IERC20 _token) external view returns (uint256) {
      return 0;
    }

    function getStakedETHBalance() external view returns (uint256) {
      return address(this).balance;
    }
}

contract RestakeManagerTest is Test {

    function test_not_minting() external {
        vm.warp(1000000000);
        address admin = address(0x1337);
        vm.label(admin, "admin");

        address user1 = address(0x1);
```

```

        RestakeManager manager_implementation = new RestakeManager();

        // RenzoOracle oracle_implementation = new RenzoOracle();

        MockRenzoOracle oracle = new MockRenzoOracle();
        MyToken ezToken = new MyToken();
        MyToken collateral1 = new MyToken();
        MockRoleManager roles = new MockRoleManager();

        RestakeManager manager = RestakeManager(address(new ERC1967Proxy(address(manager_implementation), abi.encodeWithSelector(manager_implementation.initialize.selector,
          address(roles),
          address(ezToken),
          address(oracle),
          address(0),
          address(0),
          address(0)
        ))));


        manager.addCollateralToken(collateral1);

        MockOperatorDelegator delegator = new MockOperatorDelegator();

        manager.addOperatorDelegator(IOperatorDelegator(address(delegator)));

        collateral1.mint(user1, 100*10**18);

        console.log("EZETH.totalSupply", ezToken.totalSupply());

        vm.prank(user1);
        collateral1.approve(address(manager), 100*10**18);

        vm.prank(user1);
        manager.deposit(collateral1, 100*10**18);

        console.log("EZETH.totalSupply", ezToken.totalSupply());

    }
```

```
}
```

Console output:

Listing 5

```
Logs:
  EZETH.totalSupply 0
  EZETH.totalSupply 0

          [903] MockRenzoOracle::calculateMintAmount(2000000000000000000000 [2e21], 2000000000000000000000 [2e21], 0) [staticcall]
              0
          [4805] MyToken::mint(0x0000000000000000000000000000000000000001, 0)
              emit Transfer(from: 0x0000000000000000000000000000000000000000, to: 0x0000000000000000000000000000000000000001, value: 0)
              ()
          emit Deposit(depositor: 0x0000000000000000000000000000000000000001, token: MyToken: [0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9], amount: 100000000000000000000 [1e20], ezETHMinted: 0)
           ()
        ()
     ()
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:M/D:N/Y:M/R:N/S:U (10)

Recommendation:

To address this issue, the following modifications are recommended:

1. **Adjust calculateMintAmount Logic**: Modify the logic in calculateMintAmount to account for scenarios where

_existingEzETHSupply is zero, but _currentValueInProtocol is not. This could involve implementing an additional conditional check to handle this specific case, ensuring that ezETH tokens are correctly minted for the first depositors even if the protocol already has a non-zero TVL.

2. **Fallback Minting Strategy**: Implement a fallback minting strategy in the deposit function for scenarios where ezETH.totalSupply() is zero. This strategy could mint a predetermined or calculated amount of ezETH based on the deposited value and other relevant parameters.

3. **Consider Inclusion of Minimum Supply Logic**: Introduce a minimum supply logic for the initial minting of ezETH. This approach ensures that the first depositors receive a reasonable amount of ezETH, kick-starting the protocol's economy.

4. **Robust Testing and Edge Case Analysis**: Thoroughly test the minting logic under various scenarios, including edge cases where ezETH.totalSupply() is zero. Ensure that the minting behavior aligns with the intended economic model of the protocol.

5. **Documentation and Communication**: Update the contract documentation to clearly explain the minting mechanics, especially how initial minting is handled. Communicate any changes to users and stakeholders to maintain transparency.

By making these changes, the RestakeManager contract will be able to correctly handle initial minting scenarios, ensuring fairness and consistency in the distribution of ezETH tokens to depositors.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue by returning the value added on calculateMintAmount when the _existingEzETHSupply parameter is equal to zero, in commit 66fea0d7df956fb65bf654d55cd3a1680193f147.

4.3 (HAL-03) USING TRANSFER INSTEAD OF SAFETRANSFER - MEDIUM (5.0)

Description:

It was identified that the sweepERC20() function in the DepositQueue contract uses the IERC20 interface to interact with its token parameter to transfer currencies from the contract to the feeAddress wallet. The IERC20 interface expects the transfer function to have a return value on success.

The sweepERC20() function is designed to be used with different currencies. It is important to note that the transfer functions of some tokens (e.g., USDT, BNB) do not return any values, so these tokens are incompatible with the current version of the contract.

Code Location:

The sweepERC20() function uses the IERC20 interface to interact with rewardTokenAddress:

Listing 6: contracts/Deposits/DepositQueue.sol (Line 103)

```
96     function sweepERC20(IERC20 token) external
 ↳ onlyERC20RewardsAdmin {
97         uint256 balance = IERC20(token).balanceOf(address(this));
98         if(balance > 0) {
99
100            // Sweep fees if configured
101            if(feeAddress != address(0x0) && feeBasisPoints > 0) {
102                uint256 feeAmount = balance * feeBasisPoints /
 ↳ 10000;
103                IERC20(token).transfer(feeAddress, feeAmount);
104                balance = balance - feeAmount;
105            }
106
107            token.approve(address(restakeManager), balance);
108            restakeManager.depositTokenRewardsFromProtocol(token,
 ↳ balance);
```

```
        }
    }
```

OpenZeppelin's IERC20 interface are expected to return a bool value after a successful transfer:

Listing 7: @openzeppelin/contracts/token/ERC20/IERC20.sol

```
    function transfer(address to, uint256 amount) external returns
↳ (bool);
```

Proof of Concept:

The sweepERC20() function reverts if used with tokens not having a return value (e.g., USDT):

```
├─ [43359] DepositQueue::sweepERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7)
│   ├─ [2694] RoleManager::isERC20RewardsAdmin(0x4CCeBa2d7D2B4fdcE4304d3e09a1fea9fbEb1528) [staticcall]
│   │   └─ ← true
│   ├─ [1031] 0xdAC17F958D2ee523a2206206994597C13D831ec7::balanceOf(DepositQueue: [0xA4AD4f68d0b91CFD19687c881e50f3A00242
828c]) [staticcall]
│   │   └─ ← 100000000000000 [1e14]
│   ├─ [28801] 0xdAC17F958D2ee523a2206206994597C13D831ec7::transfer(0x68E527780872cda0216Ba0d8fBD58b67a5D5e351, 100000000
0000 [1e12])
│   │   ├─ emit Transfer(from: DepositQueue: [0xA4AD4f68d0b91CFD19687c881e50f3A00242828c], to: 0x68E527780872cda0216Ba0d8
fBD58b67a5D5e351, value: 1000000000000 [1e12])
│   │   └─ ← ()
│   └─ ← "EvmError: Revert"
└─ ← "EvmError: Revert"
```

BVSS:

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

Recommendation:

It is recommended to use OpenZeppelin's SafeERC20 wrapper with the IERC20 interface to make the contracts compatible with currencies that return no value.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue in commit cea15fb1720be7617cb5632cf9a8af734e6f1e23.

4.4 (HAL-04) UNHANDLED EMPTY OPERATOR DELEGATORS LIST - LOW (2.5)

Description:

The chooseOperatorDelegatorForDeposit function in the RestakeManager contract has a potential issue when handling an empty operatorDelegators list. The function defaults to returning operatorDelegators[0] if no suitable operator delegator is found. However, if the operatorDelegators list is empty (no operator delegators have been added), attempting to access operatorDelegators[0] will result in an out-of-bounds error, causing the transaction to revert.

This issue poses a significant risk, especially if the chooseOperatorDelegatorForDeposit function is integral to the deposit process. If no operator delegators are present, users would be unable to deposit, leading to a halt in the intended functionality of the contract.

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 mitigate this issue, the following changes are recommended:

1. **Check for Empty Operator Delegators List**: Prior to accessing operatorDelegators[0], include a check to verify that the operatorDelegators list is not empty. This can be implemented using a require statement, such as `require(operatorDelegators.length > 0, "No operator delegators available");`. This validation will prevent out-of-bounds errors by ensuring that there is at least one operator delegator in the list.
2. **Validate in deposit Function**: In the deposit function, before call-

ing chooseOperatorDelegatorForDeposit, add a check to ensure that there are operator delegators available. This preemptive check will avoid calling the function when it's guaranteed to fail due to an empty list.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue by reverting if no operator is present on the list in commit 9042dca45685ad2c116303e7f87d140dad9b0e8d.

4.5 (HAL-05) DEPOSIT/WITHDRAWAL RELIANCE ON CONTRACT BALANCE - LOW (2.5)

Description:

In the deposit and completeWithdrawal functions of the OperatorDelegator contract, there is a potential issue with the way deposits are handled. The function calculates the amount to deposit based on the contract's current balance of the token (_token.balanceOf(address(this))). This approach can lead to unintended consequences when external actors transfer tokens directly to the contract's address before a deposit operation. Such transfers would artificially inflate the contract's balance, causing the subsequent deposit operation to include these externally transferred tokens, potentially exceeding the user's intended deposit amount.

This situation can be particularly problematic in scenarios where multiple users interact with the contract simultaneously, or when an external entity deliberately transfers tokens to the contract to manipulate the deposit logic. It could result in inaccurate accounting, unauthorized leveraging of deposited funds, and potential security vulnerabilities.

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 mitigate this risk, it's recommended to modify the deposit and withdrawal logic to explicitly require the amount to be deposited or withdrawn as a function argument. This change would make the deposit and withdrawal operations more predictable and secure, as it explicitly defines the amount involved in each transaction, independent of the contract's current balance. **However, to support inflationary and deflationary tokens, the argument should use the actual transferred balance to the**

operator delegator and not the requested amount. The modified approach involves:

1. **Explicit Amount Argument**: Add an argument to the deposit and withdraw functions to specify the exact amount to be deposited or withdrawn. This ensures that the user's intention is clearly communicated and adhered to by the contract. **However, to support inflationary and deflationary tokens, the argument should use the actual transferred balance to the operator delegator and not the requested amount.**
2. **Transfer Tokens to Contract**: Before calling the deposit function, the user should transfer the specified amount of tokens to the contract. This transfer should be a separate transaction, distinct from the deposit call.
3. **Check for Sufficient Balance**: In the deposit function, verify that the contract's balance of the token is at least equal to the specified deposit amount. This check ensures that the contract has received the intended amount before proceeding with the deposit operation.
4. **Use Specified Amount for Operations**: Use the explicitly provided amount in the deposit and withdraw functions for all subsequent operations, such as approvals and interactions with the strategy manager.

By adopting this approach, the contract's deposit and withdrawal functions become more robust against manipulation and unintended interactions, providing a clearer and more secure mechanism for users to manage their funds.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue by specifying the amount as an argument and using a pull pattern with an approve in commit d6ee320e1c57a776aa5500fe0227ae4cff377dee.

4.6 (HAL-06) LACK OF FEE VALIDATION - LOW (2.5)

Description:

In the DepositQueue contract, the setFeeConfig function lacks a critical validation to ensure that the feeBasisPoints are within a reasonable range, specifically less than or equal to 10000, which represents 100% of the transaction value. Without this check, an excessively high fee basis points value can lead to scenarios where the fee calculation in the receive function deducts more than the intended or reasonable amount, potentially up to the entirety of the transaction value. This oversight could lead to excessive fee deductions, draining the funds intended for staking operations.

Code Location:

The _feeBasisPoints parameter is not validated in the setFeeConfig() function:

Listing 8: contracts/Deposits/DepositQueue.sol (Line 80)

```
77     function setFeeConfig(address _feeAddress, uint256 _feeBasisPoints) external onlyRestakeManagerAdmin {
78         feeAddress = _feeAddress;
79         feeBasisPoints = _feeBasisPoints;
80     }
```

The feeBasisPoints value is used to calculate the fee amount in the DepositQueue contract:

Listing 9: contracts/Deposits/DepositQueue.sol (Line 80)

```
77     receive() external payable nonReentrant {
78         // Take protocol cut of rewards if enabled
79         if(feeAddress != address(0x0) && feeBasisPoints > 0) {
80             uint256 feeAmount = msg.value * feeBasisPoints / 10000;
```

```
            (bool success, ) = feeAddress.call{value: feeAmount}("");
            require(success, "Fee transfer failed");
        }
    }
```

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 address this issue, the following changes are recommended:

1. **Validate Fee Basis Points**: Modify the setFeeConfig function to include a check ensuring that _feeBasisPoints is less than or equal to 10000 (100%). This can be implemented using a require statement, such as require(_feeBasisPoints <= 10000, "Fee basis points too high"); . This validation will prevent setting the fees to an unreasonable level.
2. **Document Fee Limitations**: Clearly document the fee limitations in the contract's comments and user documentation. This will inform users and administrators of the contract about the acceptable range for fee basis points.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue by checking the basis points in commit 81ddedcb3ce0907d73c3bc9c7c40dd30aa22be16.

4.7 (HAL-07) MISSING ZERO ADDRESS CHECKS - LOW (2.5)

Description:

The addOperatorDelegator and addCollateralToken functions in the RestakeManager contract lacks a crucial check for null (zero) addresses. Without this validation, there is a risk that a zero address could be added to the operatorDelegators or collateralTokens lists, potentially leading to operational issues and vulnerabilities in the contract. A zero address in such a list can cause functions interacting with the list to behave unexpectedly or fail, compromising the contract's integrity and reliability.

It was identified that the several parameters in the contracts lack zero address validation.

Code Location:

src/token/EzEthToken.sol
- Line 37: initialize is missing zero address checks for _roleManager.

src/Deposits/DepositQueue.sol
- Line 56: setFeeConfig() is missing zero address checks for _feeAddress when _feeBasisPoints is greater than zero.

BVSS:

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

Recommendation:

Include a check at the beginning of the addOperatorDelegator and addCollateralToken functions to ensure that the _newOperatorDelegator and _newCollateralToken are not a zero address.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue in commit e6b0c226ceed8576a60ca53f4347058a5bcead9d.

4.8 (HAL-08) TRANSFER ALLOWED DURING PAUSE FOR MINTER/BURNER - LOW (2.5)

Description:

In the provided Solidity smart contract snippet, the _beforeTokenTransfer function is designed to control the transfer of tokens. This function allows minter and burner roles to transfer tokens even when the contract is in a paused state, using transfer and transferFrom methods. The current implementation permits minters/burners to move funds instead of strictly minting/burning. This behavior contradicts the typical pause functionality, where all transfer actions should be halted, including those by privileged roles. The function should be modified to prevent all transfers during the pause state, except for the minting (to the zero address) and burning (from the zero address), as these are the only actions allowed for minter/burner roles in a paused state.

Code Location:

Listing 10: (Line 69)

```
57     /// @dev
58     function _beforeTokenTransfer(address from, address to, uint256 amount)
59         internal virtual override
60     {
61         super._beforeTokenTransfer(from, to, amount);
62 
63         // If not paused return success
64         if (!paused) {
65             return;
66         }
67 
68         // If paused, only the minter burner can transfer tokens
69         require(roleManager.isEzETHMinterBurner(msg.sender), "Paused");
70     }
```

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 rectify this issue, the _beforeTokenTransfer function should be updated to incorporate checks that strictly enforce the pause functionality. Specifically, modify the function to:

1. Prohibit all transfers when the contract is paused, regardless of the sender's role.
2. Allow an exception for minting and burning actions. This can be achieved by checking if the from or to address is the zero address, which is a characteristic of mint and burn transactions.

Implementing these changes will ensure that the pause functionality is respected, preventing any transfer of tokens during the paused state except for the intended minting and burning actions by authorized roles. This approach also simplifies the role check, as minting and burning are inherently restricted to specific roles, and it aligns the contract behavior with standard practices regarding paused states in token contracts.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue by checking the from or to being zero when paused instead of checking the role in commit c58af47f6078e86f3852833627fe514bcf2a8d70.

4.9 (HAL-09) SYNCHRONIZED OPERATOR DELEGATOR ADDITION AND ALLOCATION SETTING - INFORMATIONAL (1.5)

Description:

In the setOperatorDelegatorAllocation function of the given contract, there is a potential issue regarding the unrestricted setting of operator delegator allocations. The function allows the Restake Manager Admin to set allocations for any IOperatorDelegator, including those not yet registered or existing in the operatorDelegators list. This could lead to situations where allocations are set for invalid or unintended delegator addresses.

The function correctly checks for null addresses and allocation basis points limits, but it lacks a validation step to ensure that the provided _operatorDelegator is a recognized and valid entity within the system. Without this check, there is a risk of misconfiguration or unintentional allocation settings, potentially leading to operational issues or exploitation.

To streamline the process and ensure consistency, it’s recommended to incorporate allocation setting within the addOperatorDelegator function and handle the allocation removal in the removeOperatorDelegator function.

BVSS:

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

Recommendation:

To mitigate this risk, the following recommendations are proposed:

1. **Validate Operator Delegator Existence**: Implement a validation mechanism to check whether the provided _operatorDelegator is an existing

and recognized entity within the system. This check should verify that the _operatorDelegator is part of the operatorDelegators list or any other relevant registry maintained by the contract.

2. **Enhance addOperatorDelegator Function**: Modify the addOperatorDelegator function to include an allocation parameter. This addition allows setting the allocation for a new operator delegator simultaneously when adding it to the system. The function signature would change to include the allocation basis points, e.g., `function addOperatorDelegator(IOperatorDelegator _newOperatorDelegator, uint256 _allocationBasisPoints)`. Ensure to validate the _allocationBasisPoints as per the existing allocation requirements.
3. **Enhance removeOperatorDelegator Function**: This function should remove the delegator from the operatorDelegators list and also clear its corresponding allocation from the operatorDelegatorAllocations mapping.

By implementing these recommendations, the contract will enforce proper validation of IOperatorDelegator entities, reducing the risk of a misconfiguration and enhancing the overall security and reliability of the allocation setting process.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue commit eb252dfcd7c3812c20bd38591c025f616a5c30c7. The code is now doing the add operation into a single function. The setOperatorDelegatorAllocation function is kept, but does check the operator existence before setting the value.

4.10 (HAL-10) MISSING DEPENDENCY INITIALIZATION - INFORMATIONAL (0.8)

Description:

It was identified that the ReentrancyGuardUpgradeable dependency used in the DepositQueue contract is not initialized in the initialize() function.

Code Location:

The ReentrancyGuardUpgradeable dependency is not initialized in the initialize() function:

Listing 11: contracts/Deposits/DepositQueue.sol

```
    function initialize(IRoleManager _roleManager) public
↳ initializer {
        require(address(_roleManager) != address(0x0),
↳ INVALID_0_INPUT);

        roleManager = _roleManager;
    }
```

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (0.8)

Recommendation:

It is recommended to call the __ReentrancyGuard_init() function in the initialize() function().

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue in commit 9f37d3a686af01b0dfc00ce4a3350ab8b68bf202.

4.11 (HAL-11) MISSING EVENTS FOR CONTRACT OPERATIONS - INFORMATIONAL (0.8)

Description:

It was identified that several admin functions from the DepositQueue and RestakeManager contracts do not emit any events. As a result, blockchain monitoring systems might not be able to timely detect suspicious behaviors related to these functions.

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:L/Y:N/R:N/S:U (0.8)

Recommendation:

Consider adding events for all important operations to help monitor the contracts and detect suspicious behavior. A monitoring system that tracks relevant events would allow the timely detection of compromised system components.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue in commit c4f17f9aff3253f1a7b93ccb6e6845f3aed841b0. All missing events were added.

4.12 (HAL-12) FOR LOOPS CAN BE GAS OPTIMIZED - INFORMATIONAL (0.0)

Description:

It was identified that several for loops employed in the RestakeManager contract can be gas optimized by the following principles:

- Unnecessary reading of the array length on each iteration wastes gas.
- A postfix (e.g. i++) operator was used to increment the i variables. It is known that, in loops, using prefix operators (e.g. ++i) costs less gas per iteration than postfix operators.
- It is also possible to further optimize loops by using unchecked loop index incrementing and decrementing.

Note that view or pure functions only cost gas if they are called from on-chain.

Code Location:

Example unoptimized for loop employed in the stakeEthInOperatorDelegator () function of the RestakeManager contract:

Listing 12: contracts/RestakeManager.sol

```
    for (uint256 i = 0; i < operatorDelegators.length; i++) {
        if (operatorDelegators[i] == operatorDelegator) {
            found = true;
            break;
        }
    }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider caching array lengths outside of loops, as long the size is not changed during the loop.

Consider using the unchecked ++i operation instead of i++ to increment the values of the uint variable inside the loop. It is noted that using unchecked operations requires particular caution to avoid overflows, and their use may impair code readability.

The following code is an example of the above recommendations:

Listing 13: For Loop Optimization

```
uint256 arrayLength = operatorDelegators.length;
for (uint256 i = 0; i < arrayLength;) {
    if (operatorDelegators[i] == operatorDelegator) {
        found = true;
        break;
    }
    unchecked { ++i; }
}
```

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue in commit 9fce5d7670bb4be283d36d1402d548b13535e9bd.

4.13 (HAL-13) USING REVERT STRINGS INSTEAD OF CUSTOM ERRORS - INFORMATIONAL (0.0)

Description:

Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. If the revert string uses strings to provide additional information about failures (e.g. `require(msg.sender == address(depositQueue), "Not Deposit Queue");`), but they are rather expensive, especially when it comes to deploying cost, and it is difficult to use dynamic information in them.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider implementing custom errors instead of reverting strings.

An example implementation of the initialization checks using custom errors:

Listing 14: Using Custom Errors

```
error AlreadyCompleted();

    function completeWithdraw(
        IStrategyManager.QueuedWithdrawal calldata withdrawal,
        uint256 middlewareTimesIndex
    ) external nonReentrant notPaused returns (uint256) {
    ...
        // require(pendingWithdrawal.completed == false, "Already
 ↳ completed");
        if (pendingWithdrawal.completed) revert AlreadyCompleted()
```

```
↳ ;
    ...
```

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue in commit 2fe42d5651424738be4067181c021414b6f3aaa1. All errors are now using custom errors instead of strings.

4.14 (HAL-14) INCOMPLETE NATSPEC DOCUMENTATION - INFORMATIONAL (0.0)

Description:

It was identified that the contracts have an incomplete **natspec** documentation. **Natspec** documentation is useful for internal developers that need to work on the project, external developers that need to integrate with the project, security professionals that have to review it but also for end users given that many chain explorers have officially integrated the support for it directly on their site.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider adding the missing **natspec** documentation, adhering to the format guideline included in Solidity documentation.

Remediation Plan:

SOLVED: The Renzo Protocol team solved this issue in commit 3a87788170fac6b29fe988fc5724441d8078f421.

REVIEW NOTES

5.1 RoleManager.sol

- Custom version of the AccessControlUpgradeable with named functions for specific roles.

5.2 EzEthToken.sol

- The initialiser does set the role manager, which will be used by onlyMinterBurner and onlyTokenAdmin to verify the access control level of the caller.
- The admin can pause the contract. This will prevent any transfer from happening when called by any address but the minter and burner.
- Minter and burner will be able to transfer tokens by using transfer and transferFrom. This means that it is possible from a minter/burner to “move” funds rather minting/burning. Probably the checks should be modified to prevent any transfer from happening if paused unless the from or to is the zero address. This allows skipping the role check, as the mint and burn can only be called by the specific role. This method will not prevent anyone from transferring to address 0, similar to burning.

5.3 OperatorDelegator.sol

- It creates a new eigenPod using the manager and verifies and stores its address to eigenPod.
- The setTokenStrategy allows the onlyOperatorDelegatorAdmin to set the strategy address for a given token. It allows overriding existing values and allows setting the strategy to the 0 address to disable deposit for it.
- The setDelegateAddress, allowed to be called only by onlyOperatorDelegatorAdmin, does set the address on the EigenLayer to delegate to by calling delegateTo on the delegationManager. This function can only be called once with a value different from

the zero address.
- The getStrategyIndex function can return a different value depending on the stakerStrategyList list, as values can be removed.

5.4 RenzoOracle.sol

- The setOracleAddress function allows setting the tokenOracleLookup mapping. The address(0) can be used to disable a lookup.
- The lookupTokenValue is verifying if the timestamp of the oracle last updated round is in a MAX_TIME_WINDOW period.

5.5 DepositQueue.sol

It does allow accumulating ETH from different sources. Once 32ETH is on the balance, the stakeEthFromQueue can be called. If anyone does send ETH directly, some fees are removed and sent to the feeAddress if set.

- The setFeeConfig allows enabling the fee on the receive function.
- The setRestakeManager can only be called by the admin.

5.6 RestakeManager.sol

- All external functionality can be paused with setPaused call by the admin.
- The addOperatorDelegator and removeOperatorDelegator functions allow manipulating the operatorDelegators list. The remove functionality does move the last list value to the removed place, **modifying the list index of any pointed value.**
- The setOperatorDelegatorAllocation can be called with an _operatorDelegator that doesn't exist yet into the operatorDelegators list.

- If the setMaxDepositTVL is set to zero, the maxDepositTVL check is skipped and not enforced.
- The getCollateralTokenIndex can return different values if one of the addresses is removed from the list. This means that indexes cannot be reused on none single-transactions.
- calculateTVLs will iterate each operator and fetch using getTokenBalanceFromStrategy each collateral token balance. The native eth tokens are stored on the last array of the operatorValues, the number of elements is stablished as collateralTokens.length + 1.
- The chooseOperatorDelegatorForDeposit function will return the first operator if only 1 is present or not found. Otherwise, it will return the first operatorDelegators whose TVL is below the threshold. If operatorDelegatorAllocations is not set, then the value is 0 and checked against tvl. Only operatorDelegators that have operatorDelegatorAllocations set will ever be used.
- The chooseOperatorDelegatorForWithdraw function will return the first operatorDelegators if on the token index there are enough funds for the ezETHValue amount. For any other operatorDelegators, the returned value should have its operatorDelegatorTVLs above the operatorDelegatorAllocations and the operatorDelegatorTokenTVLs on the index above or equal the ezETHValue amount. If none is found, the one the first on the list that holds enough ezETHValue for that token index will be used.
- The deposit function:
  - Will check using getCollateralTokenIndex that the collateral exists by fetching its index (unused return value).
  - Will verify the total TVL if set.
  - Will choose delegator based on the chooseOperatorDelegatorForDeposit return value.
  - Will mint the requested amount. Keep in mind that issue is present for inflationary and deflationary tokens. As the requested amount will not be the actual transferred amount.
- The depositETH allows depositing native ETH and getting back the ezETH amount based on the total TVL.
- The startWithdraw does:
  - Transfer the ezETH specified amount from the caller to the contract.

- Will calculate the updated TVL.

- The completeWithdraw will compute based on the parameter the withdrawalRoot. The withdrawer will be verified to be the msg.sender. Otherwise, anyone could withdraw from any root.
- The stakeEthInOperatorDelegator will send the full msg.value, which is expected to be 32 eth to the eigenPodManager. The value is then tracked and added under stakedButNotVerifiedEth for this delegator to compute the total amount for the TVL.

AUTOMATED TESTING

6.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

contracts/RestakeManager.sol

Slither results for RestakeManager.sol	
Finding	**Impact**
RestakeManager.completeWithdraw(IStrategyManager.QueuedWithdrawal,uint256) (contracts/RestakeManager.sol#525-565) uses a dangerous strict equality: - require(bool,string)(pendingWithdrawal.withdrawer == msg.sender,Not withdrawer) (contracts/RestakeManager.sol#539)	Medium
RestakeManager.completeWithdraw(IStrategyManager.QueuedWithdrawal,uint256) (contracts/RestakeManager.sol#525-565) uses a dangerous strict equality: - require(bool,string)(pendingWithdrawal.completed == false,Already completed) (contracts/RestakeManager.sol#538)	Medium

Finding	Impact
Reentrancy in RestakeManager.completeWithdraw(IStrategyManager.QueuedWithdrawal,uint256) (contracts/RestakeManager.sol#525-565): External calls: - pendingWithdrawal.operatorDelegator.completeWithdrawal(withdrawal,pendingWithdrawal.tokenToWithdraw,middlewareTimesIndex,msg.sender) (contracts/RestakeManager.sol#542-547) - ezETH.burn(address(this),pendingWithdrawal.ezETHToBurn) (contracts/RestakeManager.sol#550) State variables written after the call(s): - pendingWithdrawals[withdrawalRoot].completed = true (contracts/RestakeManager.sol#553)RestakeManagerStorageV1.pendingWithdrawals (contracts/RestakeManagerStorage.sol#37) can be used in cross function reentrancies: - RestakeManagerStorageV1.pendingWithdrawals (contracts/RestakeManagerStorage.sol#37)	Medium
RestakeManager.depositTokenRewardsFromProtocol(IERC20,uint256) (contracts/RestakeManager.sol#617-644) ignores return value by operatorDelegator.deposit(_token) (contracts/RestakeManager.sol#643)	Medium
RestakeManager.deposit(IERC20,uint256) (contracts/RestakeManager.sol#390-444) ignores return value by operatorDelegator.deposit(_collateralToken) (contracts/RestakeManager.sol#430)	Medium
RestakeManager.calculateTVLs() (contracts/RestakeManager.sol#232-292) has external calls inside a loop: operatorEthBalance = operatorDelegators[i].getStakedETHBalance() (contracts/RestakeManager.sol#273)	Low
RestakeManager.calculateTVLs() (contracts/RestakeManager.sol#232-292) has external calls inside a loop: operatorBalance = operatorDelegators[i].getTokenBalanceFromStrategy(collateralTokens[j]) (contracts/RestakeManager.sol#259-260)	Low
RestakeManager.calculateTVLs() (contracts/RestakeManager.sol#232-292) has external calls inside a loop: operatorValues[j] = renzoOracle.lookupTokenValue(collateralTokens[j],operatorBalance) (contracts/RestakeManager.sol#263-266)	Low

Finding	Impact
Reentrancy in RestakeManager.startWithdraw(uint256,IERC20) (contracts/RestakeManager.sol#453-518): External calls: - ezETH.safeTransferFrom(msg.sender,address(this),_ezEThToBurn) (contracts/RestakeManager.sol#458) - withdrawalRoot = operatorDelegator.startWithdrawal(_tokenToWithdraw,numTokensToWithdraw) (contracts/RestakeManager.sol#493-496) State variables written after the call(s): - pendingWithdrawals[withdrawalRoot] = PendingWithdrawal(_ezEThToBurn,_tokenToWithdraw,numTokensToWithdraw,msg.sender,operatorDelegator,false) (contracts/RestakeManager.sol#499-506)	Low
End of table for RestakeManager.sol	

contracts/RestakeManagerStorage.sol

Slither did not identify any vulnerabilities in the contract.

contracts/token/EzEthTokenStorage.sol

Slither did not identify any vulnerabilities in the contract.

contracts/token/EzEthToken.sol

Slither did not identify any vulnerabilities in the contract.

contracts/Delegation/OperatorDelegatorStorage.sol

Slither did not identify any vulnerabilities in the contract.

contracts/Delegation/OperatorDelegator.sol

Slither results for OperatorDelegator.sol	
Finding	**Impact**
RestakeManager.completeWithdraw(IStrategyManager.QueuedWithdrawal,uint256) (contracts/RestakeManager.sol#525-565) uses a dangerous strict equality: - require(bool,string)(pendingWithdrawal.withdrawer == msg.sender,Not withdrawer) (contracts/RestakeManager.sol#539)	Medium
RestakeManager.completeWithdraw(IStrategyManager.QueuedWithdrawal,uint256) (contracts/RestakeManager.sol#525-565) uses a dangerous strict equality: - require(bool,string)(pendingWithdrawal.completed == false,Already completed) (contracts/RestakeManager.sol#538)	Medium

Finding	Impact
Reentrancy in RestakeManager.completeWithdraw(IStrategyManager.QueuedWithdrawal,uint256) (contracts/RestakeManager.sol#525-565): External calls: - pendingWithdrawal.operatorDelegator.completeWithdrawal(withdrawal,pendingWithdrawal.tokenToWithdraw,middlewareTimesIndex,msg.sender) (contracts/RestakeManager.sol#542-547) - ezETH.burn(address(this),pendingWithdrawal.ezETHToBurn) (contracts/RestakeManager.sol#550) State variables written after the call(s): - pendingWithdrawals[withdrawalRoot].completed = true (contracts/RestakeManager.sol#553)RestakeManagerStorageV1.pendingWithdrawals (contracts/RestakeManagerStorage.sol#37) can be used in cross function reentrancies: - RestakeManagerStorageV1.pendingWithdrawals (contracts/RestakeManagerStorage.sol#37)	Medium
RestakeManager.depositTokenRewardsFromProtocol(IERC20,uint256) (contracts/RestakeManager.sol#617-644) ignores return value by operatorDelegator.deposit(_token) (contracts/RestakeManager.sol#643)	Medium
RestakeManager.deposit(IERC20,uint256) (contracts/RestakeManager.sol#390-444) ignores return value by operatorDelegator.deposit(_collateralToken) (contracts/RestakeManager.sol#430)	Medium
OperatorDelegator.stakeEth(bytes,bytes,bytes32) (contracts/Delegation/OperatorDelegator.sol#252-258) should emit an event for: - stakedButNotVerifiedEth += msg.value (contracts/Delegation/OperatorDelegator.sol#257)	Low
RestakeManager.calculateTVLs() (contracts/RestakeManager.sol#232-292) has external calls inside a loop: operatorEthBalance = operatorDelegators[i].getStakedETHBalance() (contracts/RestakeManager.sol#273)	Low
RestakeManager.calculateTVLs() (contracts/RestakeManager.sol#232-292) has external calls inside a loop: operatorBalance = operatorDelegators[i].getTokenBalanceFromStrategy(collateralTokens[j]) (contracts/RestakeManager.sol#259-260)	Low

Finding	Impact
RestakeManager.calculateTVLs() (contracts/RestakeManager.sol#232-292) has external calls inside a loop: operatorValues[j] = renzoOracle.lookupTokenValue(collateralTokens[j],operatorBalance) (contracts/RestakeManager.sol#263-266)	Low
OperatorDelegator.getStrategyIndex(IStrategy) (contracts/Delegation/OperatorDelegator.sol#153-165) has external calls inside a loop: strategyManager.stakerStrategyList(address(this),i) == _strategy (contracts/Delegation/OperatorDelegator.sol#158)	Low
Reentrancy in OperatorDelegator.verifyWithdrawalCredentials(uint64, uint40,BeaconChainProofs.ValidatorFieldsAndBalanceProofs,bytes32[]) (contracts/Delegation/OperatorDelegator.sol#263-279):External calls: - eigenPod.verifyWithdrawalCredentialsAndBalance(oracleBlockNumber, validatorIndex,proofs,validatorFields) (contracts/Delegation/OperatorDelegator.sol#269-274) State variables written after the call(s): - stakedButNotVerifiedEth -= (validatorCurrentBalanceGwei * GWEI_TO_WEI) (contracts/Delegation/OperatorDelegator.sol#278)	Low
Reentrancy in OperatorDelegator.initialize(IRoleManager,IStrategyManager,address,IDelegationManager,IEigenPodManager) (contracts/Delegation/OperatorDelegator.sol#70-96):External calls: - eigenPodManager.createPod() (contracts/Delegation/OperatorDelegator.sol#92) State variables written after the call(s): - eigenPod = IEigenPod(eigenPodManager.ownerToPod(address(this))) (contracts/Delegation/OperatorDelegator.sol#95)	Low
Reentrancy in OperatorDelegator.stakeEth(bytes,bytes,bytes32) (contracts/Delegation/OperatorDelegator.sol#252-258):External calls: - eigenPodManager.stake{value: msg.value}(pubkey,signature,depositDataRoot) (contracts/Delegation/OperatorDelegator.sol#254) State variables written after the call(s): - stakedButNotVerifiedEth += msg.value (contracts/Delegation/OperatorDelegator.sol#257)	Low

Finding	Impact
Reentrancy in RestakeManager.startWithdraw(uint256,IERC20) (contracts/RestakeManager.sol#453-518): External calls: - ezETH.safeTransferFrom(msg.sender,address(this),_ezEThToBurn) (contracts/RestakeManager.sol#458) - withdrawalRoot = operatorDelegator.startWithdrawal(_tokenToWithdraw,numTokensToWithdraw) (contracts/RestakeManager.sol#493-496) State variables written after the call(s): - pendingWithdrawals[withdrawalRoot] = PendingWithdrawal(_ezEThToBurn,_tokenToWithdraw,numTokensToWithdraw,msg.sender,operatorDelegator,false) (contracts/RestakeManager.sol#499-506)	Low
End of table for OperatorDelegator.sol	

contracts/Oracle/RenzoOracle.sol

Slither results for RenzoOracle.sol	
Finding	**Impact**
RenzoOracle.lookupTokenValue(IERC20,uint256) (contracts/Oracle/RenzoOracle.sol#63-72) has external calls inside a loop: (price,timestamp) = oracle.latestRoundData() (contracts/Oracle/RenzoOracle.sol#67)	Low
RenzoOracle.lookupTokenAmountFromValue(IERC20,uint256) (contracts/Oracle/RenzoOracle.sol#76-85) uses timestamp for comparisons Dangerous comparisons: - require(bool,string)(timestamp >= block.timestamp - MAX_TIME_WINDOW,Stale price) (contracts/Oracle/RenzoOracle.sol#81)	Low
RenzoOracle.lookupTokenValue(IERC20,uint256) (contracts/Oracle/RenzoOracle.sol#63-72) uses timestamp for comparisons Dangerous comparisons: - require(bool,string)(timestamp >= block.timestamp - MAX_TIME_WINDOW,Stale price) (contracts/Oracle/RenzoOracle.sol#68)	Low
End of table for RenzoOracle.sol	

contracts/Oracle/RenzoOracleStorage.sol

Slither did not identify any vulnerabilities in the contract.

contracts/Permissions/RoleManager.sol

Slither did not identify any vulnerabilities in the contract.

AUTOMATED TESTING

contracts/Permissions/RoleManagerStorage.sol

Slither did not identify any vulnerabilities in the contract.

contracts/Deposits/DepositQueueStorage.sol

Slither did not identify any vulnerabilities in the contract.

contracts/Deposits/DepositQueue.sol

Slither results for DepositQueue.sol	
Finding	**Impact**
DepositQueue.stakeEthFromQueue(IOperatorDelegator,bytes,bytes,bytes32) (contracts/Deposits/DepositQueue.sol#88-92) sends eth to arbitrary user Dangerous calls: - restakeManager.stakeEthInOperatorDelegator{value: 32000000000000000000}(operatorDelegator,pubkey,signature,depositDataRoot) (contracts/Deposits/DepositQueue.sol#91)	High
DepositQueue.sweepERC20(IERC20) (contracts/Deposits/DepositQueue.sol#96-110) ignores return value by IERC20(token).transfer(feeAddress,feeAmount) (contracts/Deposits/DepositQueue.sol#103)	High
DepositQueue.sweepERC20(IERC20) (contracts/Deposits/DepositQueue.sol#96-110) ignores return value by token.approve(address(restakeManager),balance) (contracts/Deposits/DepositQueue.sol#107)	Medium
DepositQueue.setFeeConfig(address,uint256)._feeAddress (contracts/Deposits/DepositQueue.sol#56) lacks a zero-check on : - feeAddress = _feeAddress (contracts/Deposits/DepositQueue.sol#57)	Low
End of table for DepositQueue.sol	

Results summary:

The findings obtained as a result of the Slither scan were reviewed. The majority of Slither findings were determined false-positives.

THANK YOU FOR CHOOSING

HALBORN