Kelp

Kelp Liquid Restaking Token (LRT) Review

Version: 2.1

December, 2023

Contents

Introduction 2
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary 3
- Findings Summary . . . 3

Detailed Findings 4

Summary of Findings 5
- Duplicate Node Delegators Not Accounted For . . . 6
- Unexpected Amount of Supported Assets Could Increase `rsETH` price . . . 7
- Potential For Slashing Event To Impact Mint Amounts . . . 8
- Reachable Arithmetic Overflow . . . 10
- Incomplete Interface Definition & Implementation . . . 11
- Potential Inconsistencies & Miscalculations With Asset Strategies . . . 12
- Inability To Remove Supported Assets . . . 13
- Potential For Inconsistent `lrtConfig` Across Contracts . . . 14
- Use of Deprecated Chainlink Function . . . 15
- Miscellaneous General Comments . . . 16

A Test Suite 20

B Vulnerability Severity Classification 22

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Kelp 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 Kelp 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 Kelp smart contracts.

Overview

The Kelp LRT (Liquid Restaking Token) project is a liquid restaking solution on Ethereum, designed to enhance the staking experience. It is a non-custodial protocol that allows users to stake their assets into and earn rewards without locking their funds, thereby maintaining liquidity.

The core components of the codebase are the following:

1. **`LRTConfig.sol`** Serves as the configuration center for the protocol. It manages the list of supported assets, their deposit limits and corresponding staking strategies.
2. **`LRTDepositPool.sol`** Handles the deposits of liquid staking tokens (LSTs) and facilitates the allocation of assets to various node delegators. It also manages the minting of `rsETH` tokens.
3. **`LRTORacle.sol`** Acts as the oracle for asset prices within the ecosystem.
4. **`NodeDelegator.sol`** Manages the delegation of assets to different strategies and transferring assets back to `LRTDepositPool.sol`.
5. **`RSETH.sol`** Represents the restaked ETH in the form of an ERC20 token.

Security Assessment Summary

This review was conducted on the files hosted on the Kelp repository and were assessed at commit 3b4e36c.

Retesting was performed on commit 2af8a04.

The list of assessed contracts is as follows.

- LRTConfig.sol
- LRTDepositPool.sol
- LRTOracle.sol
- NodeDelegator.sol
- RSETH.sol
- ChainlinkPriceOracle.sol
- LRTConfigRoleChecker.sol
- LRTConstants.sol
- UtilLib.sol
- IEigenStrategyManager.sol
- ILRTConfig.sol
- ILRTDepositPool.sol
- ILRTOracle.sol
- INodeDelegator.sol
- IPriceFetcher.sol
- IRSETH.sol
- IStrategy.sol

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities 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 all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used 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.

Findings Summary

The testing team identified a total of 10 issues during this assessment. Categorised by their severity:

- Medium: 2 issues.
- Low: 3 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Kelp 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
LRT-01	Duplicate Node Delegators Not Accounted For	Medium	Resolved
LRT-02	Unexpected Amount of Supported Assets Could Increase `rsETH` price	Medium	Resolved
LRT-03	Potential For Slashing Event To Impact Mint Amounts	Low	Closed
LRT-04	Reachable Arithmetic Overflow	Low	Closed
LRT-05	Incomplete Interface Definition & Implementation	Low	Resolved
LRT-06	Potential Inconsistencies & Miscalculations With Asset Strategies	Informational	Closed
LRT-07	Inability To Remove Supported Assets	Informational	Closed
LRT-08	Potential For Inconsistent `lrtConfig` Across Contracts	Informational	Closed
LRT-09	Use of Deprecated Chainlink Function	Informational	Resolved
LRT-10	Miscellaneous General Comments	Informational	Closed

LRT-01	Duplicate Node Delegators Not Accounted For		
Asset	`LRTDepositPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `LRTDepositPool` contract does not check for duplicate node delegator addresses when they are added to the `nodeDelegatorQueue`. This could allow the same node delegator to be added multiple times. As a result, functions that rely on this queue, such as `getAssetDistributionData()`, will return incorrect values, particularly for the `assetLyingInNDCs` calculation.

This issue can be illustrated as follows:

1. An admin mistakenly adds the same node delegator address twice to the `nodeDelegatorQueue` using `addNodeDelegatorContractToQueue()`.
2. An asset is deposited into the `LRTDepositPool`.
3. The asset is transferred to the first node delegator in the queue.
4. `getAssetDistributionData()` is called to calculate the asset distribution, including `assetLyingInNDCs`.
5. Due to the duplicate entry, the calculation for `assetLyingInNDCs` incorrectly reflects double the real amount transferred.

Recommendations

The testing team recommends implementing a check in `addNodeDelegatorContractToQueue()` to ensure a node delegator address is not already present in the `nodeDelegatorQueue` before adding it.

Resolution

A mapping `isNodeDelegator` was added to check if node delegator is already present in the queue.

This issue has been addressed in commit 5ef07df.

LRT-02	Unexpected Amount of Supported Assets Could Increase `rsETH` price		
Asset	`LRTDepositPool.sol, NodeDelegator.sol & LRTOracle.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

If, for any reason, the `LRTDepositPool` contract or the `NodeDelegator` contract, received an unexpected amount of supported assets, the `rsETH` price would increase.

The contract `LRTDepositPool` could receive intentionally (from a malicious user) or unintentionally (accidentally) an amount of supported assets without the function `depositAsset()`. Also, the contract `NodeDelegator` could also get an amount of supported assets without calling the function `transferAssetToNodeDelegator()`.

According to the function `LRTOracle.updateRSETHPrice()`, `rsETHPrice = totalETHInPool / rsEthSupply`. So, if one of the `LRTDepositPool` contracts or the `NodeDelegator` contract receives an unexpected amount of supported assets, the `rsETH` price would increase. This is because when getting unexpected amount of supported assets, no new `rsETH` tokens minted, so the `rsEthSupply` would not increase. However, the total amount of supported asset would increase and so the `totalETHInPool` would increase.

An attack scenario that can occur is that the first minter can maliciously manipulate the share price to take a profit from future user's deposits.

This can be done by first depositing the lowest possible amount of supported assets (1 wei) to the deposit pool, then transferring a large amount of assets to the pool contract directly.

This will artificially inflate the share price of `rsETH` for future depositors.

Recommendations

Consider using a system of internal accounting to track the amount of each supported asset within the project, so that any excess could be withdrawn.

A potential solution to the share inflation attack is implementing a decimal offset virtual shares and assets to the pool. See this link more details: Inflation Attacks With Virtual Shares And Assets

Alternatively an initial mint could also be used to mitigate the risk of this issue.

Resolution

Kelp DAO has elected to resolve this issue using an initial mint immediately after deployment.

LRT-03	Potential For Slashing Event To Impact Mint Amounts		
Asset	`LRTDepositPool.sol & LRTOracle.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The accuracy of the `rsETHPrice` calculated by `lrtOracle.rsETHPrice` in the `LRTDepositPool` contract is dependent on frequent updates via `LRTOracle.updateRSETHPrice()`.

There are two instances where a change in price can occur:

1. EigenLayer or staking rewards distribution
2. A mass slashing event

Currently there is no incentive for 3rd parties to call `LRTOracle.updateRSETHPrice()` due to gas costs.

The Kelp team's intended approach of updating the `rsETH` price once or twice per day, in practice, is adequate for accounting for price increases from staking and EigenLayer rewards, assuming EigenLayer rewards are distributed in a linear fashion similar to staking rewards throughout the year.

However it may not be timely enough to account for any mass slashing event.

If such an event occurs, minters will receive less than their fair share of `rsETH` until such time that `LRTOracle.updateRSETHPrice()` is called.

Recommendations

Consider automatically calling `updateRSETHPrice()` within the `getRsETHAmountToMint()` function to ensure the `rsETH` price is always current before minting.

Alternatively, monitor EigenLayer rewards distribution to ensure they follow similar distribution patterns as ETH staking rewards so that, in practice, there is no profitable incentive to frontrun any reward distribution.

If the gas trade off is deemed to be not worth while for mint. Consider calling `updateRSETHPrice()` automatically when implementing future withdraw functionality, to prevent any frontrunning and potential insolvency.

Resolution

The issue was acknowledged by the project team providing the following comment as to why the issue does not need to be fixed.

"The reason we do not update exchange rate on an asset deposit is because it increases gas cost to end user. With the slow change of LST/LRT exchange rate, we believe it is practical to rely on exchange rate update twice a day."

LRT-04	Reachable Arithmetic Overflow		
Asset	`LRTConfig.sol & LRTDepositPool.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In the `LRTDepositPool` contract, the function `getAssetCurrentLimit()` calculates the current deposit limit for an asset by subtracting the total asset deposits (obtained from `getTotalAssetDeposits()`) from the deposit limit defined in `LRTConfig.depositLimitByAsset()`. If the total deposits for an asset exceed the newly set deposit limit in `LRTConfig`, this can lead to an arithmetic underflow.

Consider the following sequence of actions demonstrating this issue:

1. A manager sets the deposit limit for an asset to a certain value, say `X`.
2. Users deposit assets, filling up this limit, so the total deposited amount equals `X`.
3. Now, the current limit as per `LRTDepositPool.getAssetCurrentLimit()` is `X - X = 0`. The manager then reduces the deposit limit for this asset to a value less than `X`, say `X/2`.
4. A call to `LRTDepositPool.getAssetCurrentLimit()` tries to compute `X/2 - X`, resulting in an arithmetic underflow.

Recommendations

Consider implementing a check in `LRTDepositPool.getAssetCurrentLimit()` to ensure that the deposit limit is not set to a value lower than the total amount already deposited for the asset.

Resolution

The issue was acknowledged by the project team.

LRT-05	Incomplete Interface Definition & Implementation		
Asset	`IRSETH.sol & RSETH.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The interface `IRSETH` does not include the definitions for several functions that are implemented in the `RSETH` contract. Specifically, it lacks definitions for `burnFrom`, `pause`, `unpause`, and `updateLRTConfig`.

Additionally, the `RSETH` contract includes a `burnFrom` function, but the corresponding `burn` function is not implemented, so there is a mismatch between the contract and its interface.

Recommendations

Ensure that the `RSETH` contract either properly implements the `IRSETH` interface or removes the interface if it is not needed. If the contract is meant to follow the `IRSETH` interface, consider implementing all functions declared in the interface to comply with the intended design.

Resolution

The `IRSETH` interface was modified according to the recommendations.

This issue has been addressed in commit 5ef07df.

LRT-06	Potential Inconsistencies & Miscalculations With Asset Strategies
Asset	`LRTConfig.sol & NodeDelegator.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `updateAssetStrategy()` function in `LRTConfig` allows changing the strategy for an asset.

In the current implementation the EigenLayer team have indicted that there will only be 1 strategy for each supported asset. However, this could change in the future, either through implementing multiple strategies per asset, or having multi asset strategies.

Any of these changes could be a breaking change for Kelp LRT.

Multiple Strategies per Asset: In this scenario, if there are assets still in the old strategy when it is changed in Kelp LRT, assets in the old strategy will not be accounted for in calculations `getAssetBalance()`. Which will in turn lead to inaccurate `rsETH` price calculations.

Multi Asset Strategies: In this scenario, there is potential for a single strategy to support multiple tokens used in Kelp LRT. If this is the case, care must be taken to ensure that this strategy must not be used for across multiple tokens in Kelp LRT. Otherwise, assets will be accounted for multiple times when looping through supported assets during `rsETH` price updates.

Recommendations

Monitor any breaking changes from EigenLayer both in terms of multiple strategies per asset or multi asset strategies.

Ensure that strategies are not modified without a proper migration process and never using the same strategy across multiple tokens respectively.

Resolution

The issue was acknowledged by the project team.

LRT-07	Inability To Remove Supported Assets
Asset	`LRTConfig.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `LRTConfig` contract allows adding new supported assets through `addNewSupportedAsset()`, but there is no functionality to remove an asset once it is added. This limitation could pose challenges in managing the list of supported assets, especially if an asset needs to be removed due to changes in the project's scope or strategy.

Recommendations

Consider adding a function to remove supported assets. This addition will enhance the contract's flexibility and adaptability to changing requirements.

Resolution

The issue was acknowledged by the project team.

LRT-08	Potential For Inconsistent `lrtConfig` Across Contracts
Asset	`RSETH.sol, LRTDepositPool.sol, LRTOracle.sol & NodeDelegator.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `RSETH` contract contains a `updateLRTConfig` function that allows updating the `lrtConfig` address. While `LRTDepositPool`, `LRTOracle`, and `NodeDelegator` also inherit a `updateLRTConfig` function from `LRTConfigRoleChecker`, there is a risk of having different `lrtConfig` addresses across these contracts if they are not updated simultaneously.

This could lead to inconsistencies, as `lrtConfig` acts as a central configuration point.

Recommendations

Consider implementing a management contract or mechanism that ensures simultaneous updates of the `lrtConfig` address across all relevant contracts.

Otherwise, ensure clear documentation of the process and implications of updating the `lrtConfig` address. Ensure that all relevant parties are aware of the need to maintain consistency across contracts.

Resolution

The issue was acknowledged by the project team.

LRT-09	Use of Deprecated Chainlink Function
Asset	`ChainlinkPriceOracle.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `ChainlinkPriceOracle` contract uses the `latestAnswer` function from the Chainlink Aggregator Interface to fetch asset prices. This method is deprecated as per the Chainlink Data Feeds API, and its continued use may lead to compatibility issues or lack of support in the future.

Recommendations

Consider replacing the deprecated `latestAnswer()` function with `latestRoundData()`.

Resolution

This issue has been addressed in commit b8b5bf6.

LRT-10	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

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

1. **Strategy Defaults to Address(0)**

   Related Asset(s): LRTConfig.sol

   Prior to `updateAssetStrategy()` being called, the strategy for a supported asset will be address(0), which will cause a revert when depositing assets.

   Ensure that strategy is updated though `updateAssetStrategy()` each time a supported asset is added after initialisation or when adding a new supported asset.

2. **Check Return Value of `depositIntoStrategy()`**

   Related Asset(s): NodeDelegator.sol

   The return value for `depositIntoStrategy()` is not checked.

   For safety reasons it is recommended to check that the number of shares returned by the function `depositIntoStrategy()` is greater than 0.

3. **Potential Reversion With Large Approvals of Non-standard ERC20**

   Related Asset(s): NodeDelegator.sol

   In the `NodeDelegator` contract, the `maxApproveToEigenStrategyManager` function approves a maximum amount (`type(uint256).max`) of an ERC20 token. While this is a common pattern for convenience, certain ERC20 tokens, like `UNI` and `COMP`, might revert when approving values higher than `type(uint96).max` . This behaviour stems from their implementation peculiarities.

   Ensure that any ERC20 tokens added to the supported list are evaluated for compatibility with large approvals.

4. **Location of `DEFAULT_ADMIN_ROLE` Definition**

   Related Asset(s): LRTConfigRoleChecker.sol

   In the `LRTConfigRoleChecker` contract, the `DEFAULT_ADMIN_ROLE` is defined as a constant with the value `0x00` . While this is a functional approach, it might be more maintainable and clearer to define such constants in a centralised location, particularly if they are shared across multiple contracts or are fundamental to the system's role-based access control.

   Consider moving the `DEFAULT_ADMIN_ROLE` constant to `LRTConstants.sol` . This can improve code maintainability and readability, especially for constants that are fundamental to the system's architecture.

5. **Prefer Use of `safeTransfer()` Over `transfer()`**

   Related Asset(s): LRTDepositPool.sol & NodeDelegator.sol

   In the `LRTDepositPool` and `NodeDelegator` contracts, using the `transfer()` and `transferFrom()` functions for ERC-20 tokens may not be the safest approach. While the `transfer()` function is a part of the ERC-20 standard, it does not always guarantee that the transfer will be successful, especially if the token contract does not follow

the ERC-20 standard perfectly. In some cases, the `transfer()` function might not revert on failure, leading to a false assumption of a successful transaction.

Consider replacing the transfer function with `safeTransfer()` and `safeTransferFrom()` from OpenZeppelin's `SafeERC20` library. This provides additional checks that ensure the transfer was successful.

6. **High Centralisation Risk**

   Related Asset(s): LRTConfig.sol

   The `LRTConfig` contract allows changing the `rsETH` address through its setter function. This capability presents a high centralisation risk, as it gives significant power to the admin.

   Thoroughly review the necessity of allowing the `rsETH` address to be updated after initialisation. If it's crucial for upgradeability or administrative purposes, clearly document the rationale and ensure robust security measures.

7. **Lack of Synchrony when Adding a New Token**

   Related Asset(s): LRTConfig.sol

   Function `getLSTToken()` and `setToken()` appear to pose as helper functions for external applications or end users to get the address of supported tokens based on `bytes32` identifier on `LRTConstants` (i.e., `R_ETH_TOKEN`, `ST_ETH_TOKEN`, and `CB_ETH_TOKEN`).

   However, there is a disconnect with the function `addNewSupportedAsset()`. In the function `initialize()`, `_setToken()` and `_addNewSupportedAsset()` are called subsequently with preset values, so the resulting values are synchronous. If a new token is added, the functions `addNewSupportedAsset()` and `setToken()` need to be called separately by different roles, allowing conflicting updates.

   Furthermore, new tokens other than the preset tokens will have no values in `LRTConstants`. This can be a consideration whether the `bytes32` identifier should be taken off `LRTConstants` so it can be flexibly added or removed.

   Consider bundling all necessary actions in one call when adding a new token to minimise potential mistakes.

8. **Zero Key Provides Default Value**

   Related Asset(s): LRTConfig.sol

   Function `_setToken()` and `_setContract()` allow the input `key` to be zero. If a value is set with zero key, then according to the current behaviour of function calling in Solidity, the getter will return the corresponding value when the user does not set any input on it (i.e., an empty byte as input).

   Make sure this behaviour is understood. Consider preventing zero key on `_setToken()` and `_setContract()` if return values with zero or empty byte input of `getLSTToken()` and `getContract()` are undesirable.

9. **Initial Value of `rsETHPrice`**

   Related Asset(s): LRTOracle.sol

   Function `updatePriceOracleFor()` must be called at least once to initialise `rsETHPrice` to a non-zero value. If the value of `rsETHPrice` is zero, then function `LRTDepositPool.depositAsset()` will revert with `division or modulo by 0`.

   Consider setting `rsETHPrice` to initial value either during initialisation or in the variable definition.

10. **Check for Nonzero Balance**

    Related Asset(s): NodeDelegator.sol

    Function `depositAssetIntoStrategy()` deposits the contract's token balance into EigenLayer's `StrategyManager` contract. This function is callable even if the balance is zero. If this occurs, the gas would be wasted.

    Consider checking that the `balance` is more than zero before proceeding.

11. **Lack of Event Emission**

(a) **Related Asset(s): NodeDelegator.sol**

In the `NodeDelegator` contract, the `transferBackToLRTDepositPool()` function, being state-changing, should ideally emit an event to ensure transparency and traceability of its actions on the blockchain.

Consider modifying the `transferBackToLRTDepositPool()` function to emit an event whenever it successfully changes the state.

(b) **Related Asset(s): LRTConfig.sol**

Function `updateAssetStrategy()` is a state-changing function. The current implementation does not emit any event.

Consider emitting an event for better traceability and visibility.

12. **Redundant Function Override**

Related Asset(s): RSETH.sol

In the `RSETH` contract, the function `updateLRTConfig()` is overridden, but it appears to have the same functionality as `updateLRTConfig()` in the `LRTConfigRoleChecker` contract, which `RSETH` inherits from.

This redundancy might be unnecessary unless there is a specific requirement for different access controls or additional functionality in `RSETH`.

If the `updateLRTConfig()` function in `RSETH` does not add any new functionality or change the access control mechanism, consider removing this override to reduce redundancy. The inherited function from `LRTConfigRoleChecker` should suffice.

13. **Gas Efficiency in Zero Address Check**

Related Asset(s): UtilLib.sol

The `UtilLib` library contains a function `checkNonZeroAddress`, used to validate that an address is not the zero address. While this is a common and necessary check in smart contracts, the current implementation using high-level Solidity may not be the most gas-efficient approach. There's an opportunity to optimise this check by using inline assembly.

Consider implementing the zero address check using Solidity inline assembly. Assembly can be more gas-efficient as it allows for lower-level manipulation of the EVM.

14. **Incorrect Parameter Description in the `initialize()` Function**

Related Asset(s): LRTConfig.sol

In the `initialize()` function of the `LRTConfig` contract, the parameter for `rsETH` is mistakenly described as `cbETH` address in the comment on line [**40**].

```
/// @param rsETH_ cbETH address
```

Correct the parameter description to accurately reflect that `rsETH_` is the `rsETH address`, not `cbETH address`. This change will improve the clarity and accuracy of the code documentation.

15. **Overlapping Getter Functions**

Related Asset(s): LRTConfig.sol

The contract `LRTConfig` has public state variables `tokenMap` and `contractMap` that automatically generate getter functions due to their `public` visibility. However, the contract also explicitly defines getter functions `getLSTToken(bytes32 tokenKey)` and `getContract(bytes32 contractKey)` which essentially serve the same purpose. This redundancy can lead to confusion and is not efficient in terms of contract design.

Consider removing the explicit getter functions `getLSTToken()` and `getContract()` to rely on the automatically generated getters for the public variables `tokenMap` and `contractMap`. This will streamline the contract, reducing redundancy and potential confusion.

Recommendations

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

Resolution

The development team's responses to the raised issues above are as follows.

1. **Strategy Defaults to Address(0)**: `depositAssetIntoStrategy()` will revert with appropriate revert message when strategy is address(0). This issue has been addressed in commit 2af8a04.

Appendix A Test Suite

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

```
Running 1 test for test/Basic.t.sol:BasicTest
[PASS] test_VariablesWork() (gas: 2246)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.29ms

Running 5 tests for test/NodeDelegator.t.sol:NodeDelegatorTest
[PASS] test_VariablesWork() (gas: 2279)
[PASS] test_depositAssetIntoStrategy_integration() (gas: 4183607)
[PASS] test_maxApproveToEigenStrategyManager() (gas: 852678)
[PASS] test_pause_unpause() (gas: 49634)
[PASS] test_transferBackToLRTDepositPool() (gas: 1804616)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 44.73ms

Running 2 tests for test/UtilLib.t.sol:UtilLibTest
[PASS] test_VariablesWork() (gas: 2246)
[PASS] test_checkNonZeroAddress(address) (runs: 10000, u: 5752, ~: 5752)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 692.49ms

Running 8 tests for test/LRTConfig.t.sol:LRTConfigTest
[PASS] testFail_updateAssetDepositLimit_lessAmountThanDeposited() (gas: 262831)
[PASS] test_VariablesWork() (gas: 2246)
[PASS] test_addNewSupportedAsset(address,uint256,uint256) (runs: 10000, u: 117340, ~: 118083)
[PASS] test_setContract(bytes32,address) (runs: 10000, u: 89284, ~: 89284)
[PASS] test_setContractAndUpdateValue(bytes32,address,address) (runs: 10000, u: 58140, ~: 58142)
[PASS] test_setRSETH(address) (runs: 10000, u: 25811, ~: 25811)
[PASS] test_setToken(bytes32,address) (runs: 10000, u: 89337, ~: 89339)
[PASS] test_updateAssetStrategy(uint8,address,address) (runs: 10000, u: 40557, ~: 40557)
Test result: ok. 8 passed; 0 failed; 0 skipped; finished in 6.20s

Running 6 tests for test/LRTOracle.t.sol:LRTOracleTest
[PASS] test_Unexpected_asset_PriceImpact_PoC() (gas: 299341)
[PASS] test_VariablesWork() (gas: 2246)
[PASS] test_getAssetPrice(uint256) (runs: 10000, u: 44552, ~: 44764)
[PASS] test_updatePriceOracleFor(address) (runs: 10000, u: 52283, ~: 52283)
[PASS] test_updateRSETHPrice(uint256) (runs: 10000, u: 568166, ~: 568166)
[PASS] test_updateRSETHPrice_zeroValue() (gas: 509663)
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 19.87s

Running 3 tests for test/ChainlinkPriceOracle.t.sol:ChainlinkPriceOracle
[PASS] test_VariablesWork() (gas: 2246)
[PASS] test_getAssetPrice(uint8) (runs: 10000, u: 47336, ~: 47336)
[PASS] test_updatePriceFeedFor(uint8,address) (runs: 10000, u: 53238, ~: 53238)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 19.87s

Running 5 tests for test/WETH9.t.sol:WETH9Test
[PASS] test_VariablesWork() (gas: 2268)
[PASS] test_deposit(uint256) (runs: 10000, u: 42645, ~: 44561)
[PASS] test_deposit_withdraw(uint256,uint256) (runs: 10000, u: 54437, ~: 55695)
[PASS] test_transfer(uint256,uint256) (runs: 10000, u: 73069, ~: 75302)
[PASS] test_transferFrom(uint256,uint256) (runs: 10000, u: 88242, ~: 90105)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 19.87s

Running 9 tests for test/RSETH.t.sol:RSETHTest
[PASS] test_VariablesWork() (gas: 2268)
[PASS] test_burnFrom_noBalance(address,uint256) (runs: 10000, u: 27841, ~: 27842)
[PASS] test_mint_burn(uint256,uint256) (runs: 10000, u: 75511, ~: 75683)
[PASS] test_mint_burnFrom(address,uint256,uint256) (runs: 10000, u: 85815, ~: 86747)
[PASS] test_mint_burnFrom_pause_unpause(address,uint256,uint256,bool,bool) (runs: 10000, u: 93765, ~: 95685)
[PASS] test_mint_notMinter(address,address,uint256) (runs: 10000, u: 52308, ~: 52312)
[PASS] test_pause_unpause() (gas: 49694)
[PASS] test_updateLRTConfig(address) (runs: 10000, u: 28581, ~: 28581)
[PASS] test_upgrade() (gas: 1399897)
Test result: ok. 9 passed; 0 failed; 0 skipped; finished in 62.46s
```

```
Running 14 tests for test/LRTDepositPool.t.sol:LRTDepositPoolTest
[PASS] testFail_addNodeDelegatorContractToQueue_sameNode_twice_PoC() (gas: 390445)
[PASS] testFail_mintWithPriceChange_updateRSETHPrice() (gas: 326485)
[PASS] testFail_mintWithStakerRewards_updateRSETHPrice() (gas: 1440228)
[PASS] test_VariablesWork() (gas: 2246)
[PASS] test_addNodeDelegatorContractToQueue(uint256) (runs: 10000, u: 4734067, ~: 4539578)
[PASS] test_assets_using_same_strategy() (gas: 639328)
[PASS] test_depositAsset(uint256) (runs: 10000, u: 439156, ~: 439158)
[PASS] test_deposit_with_node_delegator(uint256) (runs: 10000, u: 1651591, ~: 1651591)
[PASS] test_getRsETHAmountToMint_depositAsset() (gas: 519213)
[PASS] test_getRsETHAmountToMint_initial() (gas: 87749)
[PASS] test_multiple_node_delegator() (gas: 984008)
[PASS] test_pause_unpause() (gas: 49624)
[PASS] test_transferAssetToNodeDelegator() (gas: 1701151)
[PASS] test_updateMaxNodeDelegatorCount(uint256) (runs: 10000, u: 39782, ~: 39963)
Test result: ok. 14 passed; 0 failed; 0 skipped; finished in 61.17s
```

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.

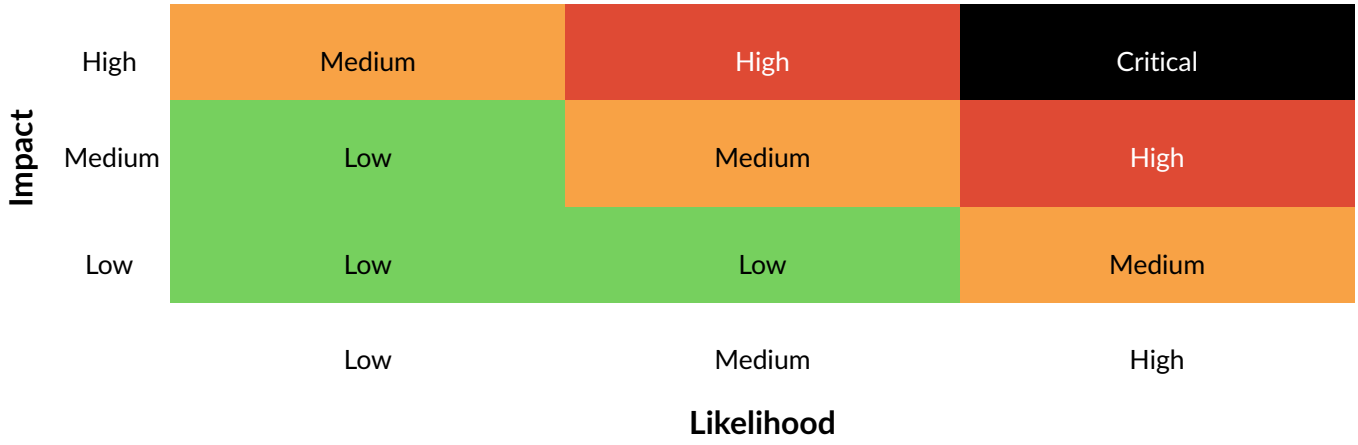

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].