KELP DAO

LRT – Smart Contract Updates

Security Assessment Report

Version: 2.0

June, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the KELP DAO smart contract updates. 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 DAO 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 DAO smart contract updates.

Overview

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

Security Assessment Summary

Scope

The scope of this time-boxed review was strictly limited to new contracts and changes to existing contracts, implemented at commits 7db0e43 since ed6fa16.

Retesting activities were performed on commit 43da3e4

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

Approach

The review was conducted on the files hosted on the KELP DAO repository, focusing on changes as per ed6fa16..7db0e43 commit diff.

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

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

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

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

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

Output for these automated tools is available upon request.

Coverage Limitations

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

Findings Summary

The testing team identified a total of 9 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 2 issues.
- Low: 2 issues.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the KELP DAO smart contract updates. 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
KLP2-01	Incorrect Accounting for `stakedButUnverifiedNativeETH`	High	Resolved
KLP2-02	*Checks-Effects-Interactions* Pattern Violations In `NodeDelegator`	Medium	Resolved
KLP2-03	No Checks on LST Price Oracles	Medium	Resolved
KLP2-04	Denial-of-Service Condition in `getETHEigenPodBalance()` Due To Overflow	Low	Resolved
KLP2-05	High Churn Rate Due To Arbitrage	Low	Closed
KLP2-06	Unimplemented `receive()` Functions in Unstaking Adapters	Informational	Closed
KLP2-07	Use of General `receive()` Functions is Discouraged	Informational	Resolved
KLP2-08	Gas Optimisations	Informational	Resolved
KLP2-09	Miscellaneous General Comments	Informational	Resolved

KLP2-01	Incorrect Accounting for `stakedButUnverifiedNativeETH`		
Asset	`NodeDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

`stakedButUnverifiedNativeETH` does not take into account an effective balance of the validator at the time of calculation, which may result in inaccurate accounting.

`stakedButUnverifiedNativeETH` in `NodeDelegator` describes the amount of `ETH` that is staked on EigenLayer, but that has not yet been verified and, as such, will not be reflected in `eigenPodManager.podOwnerShares()`. This is used to ensure that accounting is correct during this time and all of the protocol's assets are accounted for. To do this, `stakedButUnverifiedNativeETH` is incremented by 32 `ETH` when `stake32ETH()` is called, and is decremented again after verification.

NodeDelegator.sol

```
162 IEigenPodManager eigenPodManager = IEigenPodManager(lrtConfig.getContract(LRTConstants.EIGEN_POD_MANAGER));
    eigenPodManager.stake{ value: 32 ether }(pubkey, signature, depositDataRoot);
164
    // tracks staked but unverified native ETH
166 stakedButUnverifiedNativeETH += 32 ether;
```

However, `stakedButUnverifiedNativeETH` is subtracted by the effective balance of the validator, not 32 `ETH`. This presents an edge case where a validator may have an effective balance lower than 32 `ETH` during verification.

This would result in `stakedButUnverifiedNativeETH` containing some left-over `ETH`, which is counted towards the protocols funds, but is not actually owned by the protocol, resulting in inaccurate accounting and an incorrect `rsETH` price.

NodeDelegator.sol

```
226 eigenPod.verifyWithdrawalCredentials(
      oracleTimestamp, stateRootProof, validatorIndices, withdrawalCredentialProofs, validatorFields
228 );

230 uint256 totalVerifiedEthGwei = 0;
    for (uint256 i = 0; i < validatorFields.length;) {
232   // TODO: Handle case when effective balance goes below 32 eth
      // in case of validator with extra stakes, this will count 32 eth as that is max effective balance
234   uint64 validatorCurrentBalanceGwei = BeaconChainProofs.getEffectiveBalanceGwei(validatorFields[i]);
      totalVerifiedEthGwei += validatorCurrentBalanceGwei;
236   unchecked {
        ++i;
238   }
    }
240 // reduce the eth amount that is verified
    stakedButUnverifiedNativeETH -= (totalVerifiedEthGwei * LRTConstants.ONE_E_9);
```

Recommendations

Modify the accounting calculation of `stakedButUnverifiedNativeETH` to ensure it represents correct balances at all times.

Resolution

The logic in `verifyWithdrawalCredentials()` has been modified to subtract 32 ETH for every verified validator, as seen in commit bcb6193.

KLP2-02	*Checks-Effects-Interactions* Pattern Violations In `NodeDelegator`		
Asset	`NodeDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

`NodeDelegator` implements several functions that violate the *Checks-Effects-Interactions* (CEI) pattern.

Most notably in `stake32Eth()` on line [**163**], the EigenLayer contract gets control of the execution flow while the Kelp contracts are in an intermediate state.

Specifically, the `rsETH` price invariant is broken - 32 `ETH` has left the Kelp contracts but `stakedButUnverifiedNativeETH` has not yet been increased. This means that the EigenLayer contracts, or any sub-call, could call `updateRSETHPrice()` and get an incorrect `rsETH` price. Deposits and withdrawals could then be made assuming this incorrect price, leading to protocol losses:

NodeDelegator.sol

```
153 function stake32Eth(
      bytes calldata pubkey,
155   bytes calldata signature,
      bytes32 depositDataRoot
157 )
      external
159   whenNotPaused
      onlyLRTOperator
161 {
      IEigenPodManager eigenPodManager = IEigenPodManager(lrtConfig.getContract(LRTConstants.EIGEN_POD_MANAGER));
163   eigenPodManager.stake{ value: 32 ether }(pubkey, signature, depositDataRoot);

165   // tracks staked but unverified native ETH
      stakedButUnverifiedNativeETH += 32 ether;
167
      emit ETHStaked(pubkey, 32 ether);
169 }
```

Other functions where CEI violations occur:

- `stake32EthValidated()` on line [**196**]
- `verifyWithdrawalCredentials()` on line [**226**]
- `completeUnstaking()` on line [**345**]

Recommendations

Restructure the functions in question to follow the *Checks-Effects-Interactions* pattern.

Resolution

The development team has restructured the code where relevant, as seen in 0534d17.

KLP2-03	No Checks on LST Price Oracles		
Asset	`contracts/oracles/*`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

No checks are performed on Liquid Staking Token (LST) rates returned by LST price oracles.

SwETHPriceOracles.sol

```
34 function getAssetPrice(address asset) external view returns (uint256) {
   if (asset != swETHAddress) {
36   revert InvalidAsset();
   }
38
   return ISwETH(swETHAddress).getRate();
40 }
```

The LST price oracles are responsible for providing the price of an LST against `ETH`. To do this, the LST's `getRate()` function is called. However, no checks are performed on the returned rate, which may expose the Kelp protocol to considerable integration risk, especially considering that some of the LSTs are upgradeable (such as `swETH`) and the repricing of the LSTs is often controlled by an EOA. Fully trusting the `getRate()` function of an LST significantly increases the attack surface of the Kelp protocol.

Several basic measures can be taken to mitigate this risk:

- A reasonable upper and lower bound can be placed on the returned rate;
- The function call can be wrapped in a `try-catch` block to prevent forced reverts;
- This could be combined with a fallback value which is used in case of revert, such as the previous rate.

Recommendations

Ensure the above comments are understood and consider implementing the measures outlined above.

Resolution

The development team has mitigated the issue regarding rate manipulation by placing guards on the rate of change of the `rsETH` price in `LRTOracle.sol`. The development team has opted to close the aspect regarding forced reverts. These changes can be seen in PR #17.

KLP2-04	Denial-of-Service Condition in `getETHEigenPodBalance()` Due To Overflow		
Asset	`NodeDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

There is an edge case in `getETHEigenPodBalance()` in `NodeDelegator.sol` where `nativeEthShares` is negative and `-nativeEthShares > stakedButUnverifiedNativeETH`.

This would cause an overflow on line [**530**] and revert. As a result, several functions could become unavailable for execution, causing a Denial-of-Service (DoS) condition. Most notably, `_beforeDeposit()` and `updateRSETHPrice()` would both no longer be executable.

NodeDelegator.sol

```
529 return nativeEthShares < 0
    ? stakedButUnverifiedNativeETH - uint256(-nativeEthShares)
531 : stakedButUnverifiedNativeETH + uint256(nativeEthShares);
```

Since it is unlikely for this edge case to occur, the testing team rates this issue as low likelihood.

Recommendations

Add extra logic to handle the case where `-nativeEthShares > stakedButUnverifiedNativeETH`.

Resolution

The development team has fixed the issue as seen in commit 03551e7.

KLP2-05	High Churn Rate Due To Arbitrage		
Asset	`contracts/*`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The redemption rates are used to price assets. This has shown to not always equal the market price as LSTs can trade both above and below their redemption rate for extended periods of time.

This means that the `rsETH` protocol can be used for arbitrages: a below peg asset can be deposited at peg rate and can be withdrawn for any other asset in order to turn a profit.

Such market conditions would cause a lot of churn, assets would constantly be deposited and withdrawn. In turn, this may lower capital efficiency since assets would have to be deposited and withdrawn from EigenLayer instead of earning rewards.

Recommendations

One way to mitigate this partially is by charging a fee to deposit and withdraw. This would discourage small arbitrages by making them unprofitable. As depositing and withdrawing is currently free, even small and temporary depegs may trigger arbitrages.

Another mitigation would be to use the currently implemented deposit limits. If an asset were to have a larger depeg event, its deposit limits could be lowered to limit the arbitrage possibilities. This would also defend the value of `rsETH` and ensure it does not drop too much with the depegged asset.

Resolution

The development team has opted the above issue with the following statement:

> *"We have a minimum withdrawal delay of 7 days, which should discourage this. On top of that we are going to charge a fee soon."*

KLP2-06	Unimplemented `receive()` Functions in Unstaking Adapters
Asset	`UnstakeStETH.sol`, `UnstakeSwETH.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The unstaking adapters `UnstakeStETH` and `UnstakeSwETH` do not implement the `receive()` and `onERC721Received()` functions that are required to unstake `stETH` and `swETH`.

This is not an issue currently as the inheriting contract `LRTConverter` implements these. However, if these contracts are used elsewhere in the future, this could lead to significant loss of funds.

Recommendations

Consider implementing the required `receive()` functions in the unstaking contracts themselves. Alternatively, add a comment in `UnstakeStETH` and `UnstakeSwETH` that mentions these functions must be implemented in the inheriting contracts.

Resolution

The development team has opted to close this issue.

KLP2-07	Use of General `receive()` Functions is Discouraged
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In several locations, general `receive()` functions are used. It is recommended to use more specific receive functions.

For example, instead of:

```
receive() external payable {}
(bool sent,) = payable(withdrawalManager).call{ value: amount }("")
```

it is recommended to use:

```
function receiveFromUnstakingVault() external payable override {}
withdrawalManager.receiveFromUnstakingVault{value: amount}()
```

Some of the advantages to these more specific functions are:

- More readable and verbose. It shows who the intended sender is and what the intent of the transfer is;
- Avoids accidental ETH transfers from users. This is especially important for contracts such as `LRTDepositPool.sol`;
- Extra access control could potentially be placed on these functions. For example: `msg.sender` must equal `unstakingVault` in `receiveFromUnstakingVault()`.

Certain `receive{}` functions can of course not be replaced. For example, EigenLayer will send `ETH` to the `receive()` function.

Recommendations

Review code in question and make alterations as deemed applicable.

Resolution

The development team has fixed the issue by implementing specific receive functions, as seen in 9248e7b.

KLP2-08	Gas Optimisations
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Some areas of the protocol could be altered to save gas:

1. `_unlockWithdrawalRequests()` in `LRTWithdrawalManager` reads `nextLockedNonce[asset]` several times. In order to save gas, `nextLockedNonce[asset]` could be cached in a memory variable.
2. Similarly, `nextUnusedNonce[asset]` in `_addUserWithdrawalRequest()` could be cached.
3. Use `==` instead of `<=` in the function `NodeDelegator._sendRewardsToRewardReceiver()` in line [**509**].
4. `EthXPriceOracle.getAssetPrice()` performs two external calls every time it is called. This is to ensure that the given argument `asset` equals the expected address of `ETHx`. Consider performing these calls during initialisation and caching the address in storage to save gas.

```
function getAssetPrice(address asset) external view returns (uint256) {
  address staderConfigProxyAddress = IETHXStakePoolsManager(ethXStakePoolsManagerProxyAddress).staderConfig();

  if (asset != IStaderConfig(staderConfigProxyAddress).getETHxToken()) {
    revert InvalidAsset();
  }

  return IETHXStakePoolsManager(ethXStakePoolsManagerProxyAddress).getExchangeRate();
}
```

Recommendations

Review code in question and make alterations as deemed applicable.

Resolution

The development team has implemented the above suggestions in commits 3317d36, 3a96c73, d0115d2 and 0051020 respectively.

KLP2-09	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **Unused Variables**
   Related Asset(s): LRTConverter.sol
   - `conversionLimit` in `LRTConverter` and its associated functionality is unused and can be removed.
   - `processedWithdrawalRoots` in `LRTConverter` is unused and can be removed. If the storage layout must be maintained for later upgrades, consider renaming the variable (e.g.: `_legacyProcessedWithdrawalRoots`) instead to increase readability.
2. **Inconsistent Function Name**
   Related Asset(s): LRTConverter.sol, LRTWithdrawalManager.sol
   - The functions `claimStEth()` and `claimSwEth()` are not intended for claiming `stETH` and `swETH`, but instead for claiming `ETH`.
   - `setMinAmountToWithdraw()` seems to indicate that it would set the minimum amount of an asset a user is allowed to withdraw. However, it actually sets the minimum amount of `rsETH` that can be withdrawn.

   Consider renaming these functions to better describe their functionalities.
3. **Incorrect Contract Documentation**
   Related Asset(s): LRTConverter.sol
   The contract documentation

   ```
   /// @title LRTConverter - Converts eigenlayer deployed LSTs to rsETH
   /// @notice Handles eigenlayer deposited LSTs to rsETH conversion
   ```

   does not match the contract functionalities.
   Consider revising the documentation to better describe the functionalities of the contract.
4. **Duplicate Definition**
   Related Asset(s): FeeReceiver.sol
   `FeeReceiver` defines a constant `MANAGER` that is equal to the one defined in `LRTConstants.sol`. Consider deleting the definition in `FeeReceiver.sol` and referring to the one in `LRTConstants.sol` to ensure consistency.

Recommendations

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

Resolution

The development team has fixed these issues in commits 1b968ee and 386862b.

Appendix A Test Suite

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

```
Ran 1 test for test/KELP.t.sol:KelpTest
[PASS] test_initialize() (gas: 17965)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.65ms (45.21µs CPU time)

Ran 1 test for test/OneETHPriceOracle.t.sol:OneETHPriceOracleTest
[PASS] test_getAssetPrice() (gas: 14814)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.00ms (95.13µs CPU time)

Ran 3 tests for test/SwETHPriceOracle.t.sol:SwETHPriceOracleTest
[PASS] test_getAssetPrice() (gas: 20026)
[PASS] test_getAssetPrice_invalidAsset() (gas: 17769)
[PASS] test_initialize() (gas: 14881)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 9.25ms (81.96µs CPU time)

Ran 5 tests for test/RSETHPriceFeed.t.sol:RSETHPriceFeedTest
[PASS] test_decimals() (gas: 10531)
[PASS] test_getRoundData() (gas: 15224)
[PASS] test_initialize() (gas: 11556)
[PASS] test_latestRoundData() (gas: 14816)
[PASS] test_version() (gas: 10525)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 9.93ms (588.58µs CPU time)

Ran 3 tests for test/SfrxETHPriceOracle.t.sol:SfrxETHPriceOracleTest
[PASS] test_getAssetPrice() (gas: 19949)
[PASS] test_getAssetPrice_invalidAsset() (gas: 17703)
[PASS] test_initialize() (gas: 14892)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 9.54ms (82.04µs CPU time)

Ran 17 tests for test/LRTConfig.t.sol:LRTConfigTest
[PASS] test_addNewSupportedAsset() (gas: 97247)
[PASS] test_addNewSupportedAsset_existingAsset() (gas: 96636)
[PASS] test_addNewSupportedAsset_onlyDefaultAdmin() (gas: 49864)
[PASS] test_initialize() (gas: 37234)
[PASS] test_setContract() (gas: 44308)
[PASS] test_setContract_ValueAlreadyInUse() (gas: 23058)
[PASS] test_setContract_onlyDefaultAdmin() (gas: 47840)
[PASS] test_setRSETH() (gas: 25959)
[PASS] test_setRSETH_onlyDefaultAdmin() (gas: 47686)
[PASS] test_setToken() (gas: 44208)
[PASS] test_setToken_ValueAlreadyInUse() (gas: 22980)
[PASS] test_setToken_onlyDefaultAdmin() (gas: 47797)
[PASS] test_updateAssetDepositLimit() (gas: 32924)
[PASS] test_updateAssetDepositLimit_onlyManager() (gas: 49874)
[PASS] test_updateAssetDepositLimit_onlySupportedAssets() (gas: 22947)
[PASS] test_updateAssetStrategy() (gas: 47197)
[PASS] test_updateAssetStrategy_onylDefaultAdmin() (gas: 49934)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 10.78ms (948.46µs CPU time)

Ran 13 tests for test/FeeReceiver.t.sol:FeeReceiverTest
[PASS] test_initialize() (gas: 38623)
[PASS] test_sendFunds() (gas: 78805)
[PASS] test_sendFunds_depositPoolFail() (gas: 72875)
[PASS] test_sendFunds_treasuryFail() (gas: 58053)
[PASS] test_setDepositPool() (gas: 28648)
[PASS] test_setDepositPool_onlyManager() (gas: 49387)
[PASS] test_setDepositPool_zeroAddress() (gas: 20684)
[PASS] test_setProtocolFeePercentage() (gas: 26629)
[PASS] test_setProtocolFeePercentage_onlyManager() (gas: 47620)
[PASS] test_setProtocolFeePercentage_zeroFee() (gas: 20532)
[PASS] test_setProtocolTreasury() (gas: 28691)
[PASS] test_setProtocolTreasury_onlyManager() (gas: 49388)
```

```
[PASS] test_setProtocolTreasury_zeroAddress() (gas: 20618)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 11.94ms (1.89ms CPU time)

Ran 3 tests for test/RETHPriceOracle.t.sol:RETHPriceOracleTest
[PASS] test_getAssetPrice() (gas: 19971)
[PASS] test_getAssetPrice_invalidAsset() (gas: 17725)
[PASS] test_initialize() (gas: 14914)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 2.94ms (84.75µs CPU time)

Ran 4 tests for test/UtilLib.t.sol:UtilLibTest
[PASS] test_checkNonZeroAddress() (gas: 5629)
[PASS] test_checkNonZeroAddress_zero() (gas: 8460)
[PASS] test_getMax() (gas: 6637)
[PASS] test_getMin() (gas: 6550)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 6.00ms (208.96µs CPU time)

Ran 3 tests for test/EthXPriceOracle.t.sol:EthXPriceOracleTest
[PASS] test_getAssetPrice() (gas: 22069)
[PASS] test_getAssetPrice_invalidAsset() (gas: 20204)
[PASS] test_initialize() (gas: 14881)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 22.00ms (431.58µs CPU time)

Ran 18 tests for test/NodeDelegator.t.sol:NodeDelegatorTest
[PASS] testFail_receive_send() (gas: 17129)
[PASS] testFail_receive_transfer() (gas: 16980)
[PASS] test_activateRestaking_onlyManager() (gas: 30504)
[PASS] test_createEigenPod() (gas: 243349)
[PASS] test_delegateTo() (gas: 70897)
[PASS] test_delegateTo_onlyManager() (gas: 31768)
[PASS] test_depositAssetIntoStrategy() (gas: 231015)
[PASS] test_initiateNativeEthWithdrawBeforeRestaking_claimNativeEthWithdraw_transferToLRTUnstakingVault() (gas: 48672008)
[PASS] test_initiateUnstaking() (gas: 497920)
[PASS] test_maxApproveToEigenStrategyManager() (gas: 606156)
[PASS] test_receive_call() (gas: 61172)
[PASS] test_sendETHFromDepositPoolToNDC() (gas: 42576)
[PASS] test_stake32Eth() (gas: 303265)
[PASS] test_stake32EthValidated() (gas: 367825)
[PASS] test_transferBackToLRTDepositPool_ETH() (gas: 64519)
[PASS] test_transferBackToLRTDepositPool_asset() (gas: 113910)
[PASS] test_transferETHToLRTUnstakingVault() (gas: 62480)
[PASS] test_upgrade_ndc() (gas: 4330649)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 90.69ms (84.80ms CPU time)

Ran 17 tests for test/LRTConverter.t.sol:LRTConverterTest
[PASS] test_addConvertableAsset_removeConvertableAsset_happyPath(address) (runs: 1001, µ: 48003, ~: 47988)
[PASS] test_claimStETH() (gas: 60580)
[PASS] test_claimStETH_onlyOperator() (gas: 30632)
[PASS] test_claimSwEth() (gas: 56924)
[PASS] test_claimSwEth_onlyOperator() (gas: 30507)
[PASS] test_initialize() (gas: 17122)
[PASS] test_initialize2() (gas: 34489)
[PASS] test_initialize2_InitializeAgain() (gas: 27059)
[PASS] test_removeConvertableAsset_happyPath(address) (runs: 1001, µ: 39092, ~: 39092)
[PASS] test_swapEthToAsset_happyPath() (gas: 3720229)
[PASS] test_transferAssetFromDepositPool() (gas: 200912)
[PASS] test_transferAssetFromDepositPool_NotFromManager() (gas: 146025)
[PASS] test_transferAssetFromDepositPool_UnsupportedAsset() (gas: 116181)
[PASS] test_unstakeStETH() (gas: 73217)
[PASS] test_unstakeStETH_onlyOperator() (gas: 30551)
[PASS] test_unstakeSwETH_onlyOperator() (gas: 30530)
[PASS] test_unstakeSwEth() (gas: 71005)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 630.10ms (167.19ms CPU time)

Ran 11 tests for test/KelpDepositPool.t.sol:KelpDepositPoolTest
[PASS] test_getReward() (gas: 305517)
[PASS] test_initialize() (gas: 30289)
[PASS] test_notifyRewardAmount() (gas: 144892)
[PASS] test_notifyRewardAmount_onlyAdmin() (gas: 19839)
[PASS] test_setRewardsDuration() (gas: 26407)
[PASS] test_setRewardsDuration_durationNotFinished() (gas: 135810)
```

```
[PASS] test_setRewardsDuration_onlyAdmin() (gas: 17817)
[PASS] test_stake() (gas: 259950)
[PASS] test_stake_nonZero() (gas: 166130)
[PASS] test_withdraw() (gas: 226633)
[PASS] test_withdraw_nonZero() (gas: 256712)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 1.02s (1.14ms CPU time)

Ran 7 tests for test/LRTUnstakingVault.t.sol:LRTUnstakingVaultTest
[PASS] testFail_receive_send_unstakingVault(uint256) (runs: 1001, μ: 33580, ~: 34132)
[PASS] testFail_receive_transfer_unstakingVault(uint256) (runs: 1001, μ: 17512, ~: 17787)
[PASS] test_addSharesUnstaking_happyPath(uint8,address,uint256) (runs: 1001, μ: 352072, ~: 353059)
[PASS] test_addSharesUnstaking_notLRTNodeDelegator(uint8,address,uint256) (runs: 1001, μ: 56954, ~: 57190)
[PASS] test_receive_call_unstakingVault(uint256) (runs: 1001, μ: 24548, ~: 24810)
[PASS] test_redeem_eth(uint256) (runs: 1001, μ: 55898, ~: 55622)
[PASS] test_redeem_token(uint256) (runs: 1001, μ: 575548, ~: 575352)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 878.17ms (963.87ms CPU time)

Ran 24 tests for test/LRTDepositPool.t.sol:LRTDepositPoolTest
[PASS] testFail_depositAsset_inflation_noProtection() (gas: 419213)
[PASS] test_addNodeDelegatorContractToQueue(uint256) (runs: 1000, μ: 10845892, ~: 10697095)
[PASS] test_addNodeDelegatorContractToQueue_duplicate() (gas: 113883)
[PASS] test_addNodeDelegatorContractToQueue_maximumLimitReached(address[]) (runs: 1000, μ: 62089, ~: 61954)
[PASS] test_depositAsset(uint256) (runs: 1000, μ: 448465, ~: 448465)
[PASS] test_depositAsset_inflation_protected() (gas: 403474)
[PASS] test_depositETH(uint256) (runs: 1000, μ: 253788, ~: 253788)
[PASS] test_deposit_with_node_delegator(uint256) (runs: 1000, μ: 1369219, ~: 1369219)
[PASS] test_getRsETHAmountToMint(uint8,uint256,uint256) (runs: 1001, μ: 70371, ~: 70482)
[PASS] test_getRsETHAmountToMint_depositAsset() (gas: 544798)
[PASS] test_getRsETHAmountToMint_initial() (gas: 83422)
[PASS] test_pause_unpause() (gas: 50841)
[PASS] test_removeManyNodeDelegatorContractsFromQueue() (gas: 402000)
[PASS] test_removeNodeDelegatorContractFromQueue() (gas: 361007)
[PASS] test_removeNodeDelegatorContractFromQueue_nonExistence() (gas: 309370)
[PASS] test_setMinAmountToDeposit(uint256) (runs: 1001, μ: 56345, ~: 57081)
[PASS] test_swapETHForAssetWithinDepositPool(uint256) (runs: 1000, μ: 315920, ~: 315920)
[PASS] test_transferAssetToLRTUnstakingVault() (gas: 89651)
[PASS] test_transferAssetToLRTUnstakingVault_OnlyManager() (gas: 35860)
[PASS] test_transferAssetToNodeDelegator() (gas: 1417882)
[PASS] test_transferETHToLRTUnstakingVault() (gas: 57726)
[PASS] test_transferETHToLRTUnstakingVault_OnlyManager() (gas: 35594)
[PASS] test_transferETHToNodeDelegator() (gas: 665066)
[PASS] test_updateMaxNodeDelegatorLimit(uint256) (runs: 1001, μ: 41908, ~: 42119)
Suite result: ok. 24 passed; 0 failed; 0 skipped; finished in 2.21s (4.22s CPU time)

Ran 17 tests for test/LRTWithdrawalManager.t.sol:LRTWithdrawalManagerTest
[PASS] test_initiateWithdrawal_exceedAmount(uint8,uint256,uint256) (runs: 1001, μ: 497986, ~: 501029)
[PASS] test_initiateWithdrawal_exceedAmountToWithdraw() (gas: 1559848)
[PASS] test_initiateWithdrawal_exceedsDepositAmount(uint8,uint256,uint256) (runs: 1001, μ: 328925, ~: 329170)
[PASS] test_initiateWithdrawal_happyPath(uint8,uint256,uint256) (runs: 1001, μ: 501894, ~: 504595)
[PASS] test_initiateWithdrawal_invalidAmountToWithdraw(uint8,uint256,uint256) (runs: 1001, μ: 319847, ~: 319922)
[PASS] test_initiateWithdrawal_noApproveRsETH(uint8,uint256,uint256) (runs: 1001, μ: 306082, ~: 306305)
[PASS] test_initiateWithdrawal_unlockQueue_completeWithdrawal_eth_happyPath() (gas: 48961008)
[PASS] test_initiateWithdrawal_unlockQueue_completeWithdrawal_happyPath(uint8) (runs: 1001, μ: 4049046, ~: 4049084)
[PASS] test_pause_paused() (gas: 61704)
[PASS] test_pause_unpause_happyPath() (gas: 49177)
[PASS] test_receive(uint256) (runs: 1001, μ: 22649, ~: 22870)
[PASS] test_setMinAmountToWithdraw(uint256,address) (runs: 1001, μ: 55772, ~: 56548)
[PASS] test_setWithdrawalDelayBlocks_happyPath(uint256) (runs: 1001, μ: 42606, ~: 42342)
[PASS] test_setWithdrawalDelayBlocks_tooSmall(uint256) (runs: 1001, μ: 35679, ~: 35912)
[PASS] test_unlockQueue_emptyUnstakingVault(uint8,uint256,uint256,uint256) (runs: 1001, μ: 107723, ~: 107776)
[PASS] test_unlockQueue_noPendingWithdrawals(uint8,uint256,uint256,uint256) (runs: 1001, μ: 333023, ~: 332836)
[PASS] test_unpause_notPaused() (gas: 33522)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 5.18s (8.26s CPU time)

Ran 16 test suites in 5.20s (10.11s CPU time): 147 tests passed, 0 failed, 0 skipped (147 total tests)
```

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.

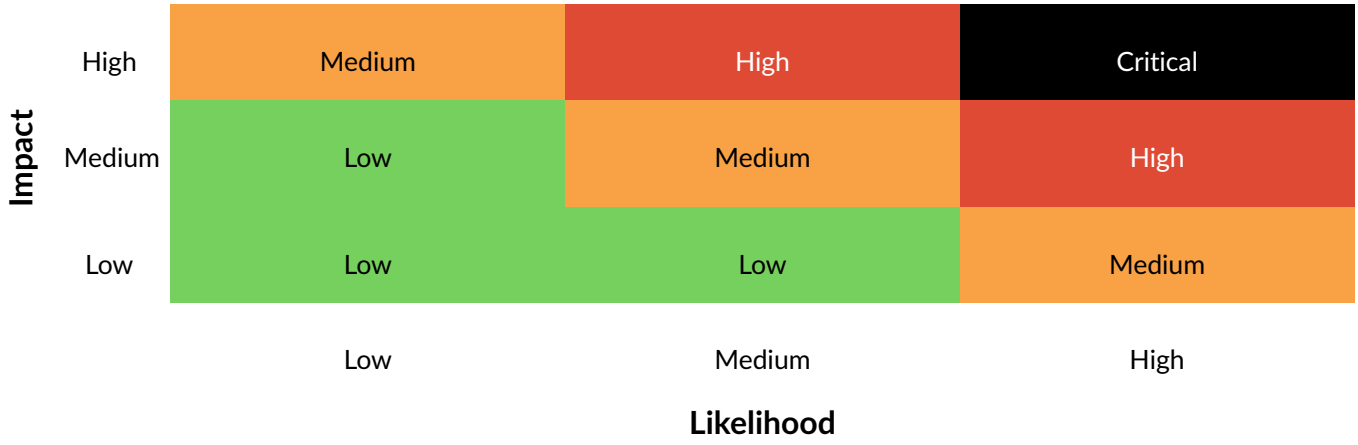

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].