Kelp DAO

PEPE Upgrade Integration

Security Assessment Report

Version: 2.0

September, 2024

Contents

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

Security Assessment Summary **3**
- Scope . . . 3
- Approach . . . 3
- Coverage Limitations . . . 3
- Findings Summary . . . 3

Detailed Findings **5**

Summary of Findings **6**
- Malicious Validator Front-Running Attack . . . 7
- Lost Funds After EigenLayer Operator-Initiated Undelegations . . . 9
- Frozen Withdrawal Due To Wrong ETH Address In `completeUnstaking()` . . . 11
- Incorrect `getETHEigenPodBalance()` Calculation Can Overinflate TVL . . . 13
- `unlockQueue()` is Susceptible To Asset Price Manipulation . . . 15
- `FeeReceiver` Balance Is Not Included In TVL . . . 16
- Incorrect `stakedButUnverifiedNativeETH` Accounting Can Overinflate TVL . . . 17
- `incrementExtraStakeToReceive()` Can Block Verifying Withdrawal Credentials . . . 19
- `unlockQueue()` Potentially Uses Outdated `rsETH` Price . . . 21
- Lack Of Precision In `pricePercentageLimit` . . . 22
- `LRTOracle` Does Not Check Decimals . . . 23
- Miscellaneous General Comments . . . 24

A Test Suite **26**

B Vulnerability Severity Classification **27**

Introduction

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

Overview

Kelp is a liquid restaking protocol built on top of EigenLayer. It allows users to deposit ETH or other EigenLayer-supported tokens to receive `rsETH`, Kelp's liquid restaking token.

This security assessment focused on upgrades to Kelp's smart contracts required to support EigenLayer's `EigenPod` PEPE upgrade.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the KelpDAO-contracts repository.

The scope of this time-boxed review was strictly limited to the diff between commits 43da3e4 and db8ae4e.

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

Approach

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`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the 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 12 issues during this assessment. Categorised by their severity:

- High: 3 issues.
- Medium: 5 issues.

- Low: 1 issue.
- Informational: 3 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Kelp DAO smart contracts in scope. 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	Malicious Validator Front-Running Attack	High	Resolved
KLP2-02	Lost Funds After EigenLayer Operator-Initiated Undelegations	High	Resolved
KLP2-03	Frozen Withdrawal Due To Wrong ETH Address In `completeUnstaking()`	High	Resolved
KLP2-04	Incorrect `getETHEigenPodBalance()` Calculation Can Overinflate TVL	Medium	Resolved
KLP2-05	`unlockQueue()` is Susceptible To Asset Price Manipulation	Medium	Resolved
KLP2-06	`FeeReceiver` Balance Is Not Included In TVL	Medium	Closed
KLP2-07	Incorrect `stakedButUnverifiedNativeETH` Accounting Can Overinflate TVL	Medium	Closed
KLP2-08	`incrementExtraStakeToReceive()` Can Block Verifying Withdrawal Credentials	Medium	Resolved
KLP2-09	`unlockQueue()` Potentially Uses Outdated `rsETH` Price	Low	Closed
KLP2-10	Lack Of Precision In `pricePercentageLimit`	Informational	Closed
KLP2-11	`LRTOracle` Does Not Check Decimals	Informational	Resolved
KLP2-12	Miscellaneous General Comments	Informational	Resolved

KLP2-01	Malicious Validator Front-Running Attack		
Asset	`NodeDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The Kelp team currently stakes validators on the beacon chain using the `stake32Eth()` function, which is vulnerable to front-running attacks that allow the node operator to steal the staked ETH from Kelp.

Staking protocols, such as Kelp, that rely on third-party node operators are vulnerable to a front-running attack, which allows the node operator to steal the staked ETH from Kelp.

The attack originates from the fact that the withdrawal credentials contained within the deposit data are ignored if the deposit is a top-up for an existing validator on the beacon chain. This can be seen in the consensus specs in the `apply_deposit()` function:

```
def apply_deposit(state: BeaconState,
                  pubkey: BLSPubkey,
                  withdrawal_credentials: Bytes32,
                  amount: uint64,
                  signature: BLSSignature) -> None:
    validator_pubkeys = [v.pubkey for v in state.validators]
    if pubkey not in validator_pubkeys:
        # Verify the deposit signature (proof of possession) which is not checked by the deposit contract
        deposit_message = DepositMessage(
            pubkey=pubkey,
            withdrawal_credentials=withdrawal_credentials,
            amount=amount,
        )
        domain = compute_domain(DOMAIN_DEPOSIT)  # Fork-agnostic domain since deposits are valid across forks
        signing_root = compute_signing_root(deposit_message, domain)
        if bls.Verify(pubkey, signing_root, signature):
            add_validator_to_registry(state, pubkey, withdrawal_credentials, amount)
    else:
        # Increase balance by deposit amount
        # @audit `withdrawal_credentials` are not checked
        index = ValidatorIndex(validator_pubkeys.index(pubkey))
        increase_balance(state, index, amount)
```

Since the withdrawal credentials are not checked for existing validators, a malicious node operator can add the validator to the beacon chain themselves, supplying their own address as the withdrawal credentials with a small 1 ETH deposit.

An LRT operator who stakes 32 ETH for this validator will have that ETH stolen since the validator's withdrawal credentials will be set to the node operator's address instead of the EigenPod.

The `NodeDelegator::stake32EthValidated()` function checks the `expectedDepositRoot` to prevent front-running attacks in the same block as shown below:

```
bytes32 actualDepositRoot = depositContract.get_deposit_root();
if (expectedDepositRoot != actualDepositRoot) {
    revert InvalidDepositRoot(expectedDepositRoot, actualDepositRoot);
}
```

However, a malicious node operator can still add the validator to the beacon chain many blocks in advance to perform the attack, as offchain checks are not performed to ensure that there are no previous deposits to the same validator public key.

Recommendations

In conjunction with using the `stake32EthValidated()` function, perform offchain checks to ensure that there have been no previous deposits to the same validator public key on the beacon chain, such that it is verified that the validator does not exist on the beacon chain before the `NodeDelegator` deposit is processed.

To check that the validator does not already exist on the beacon chain, query the beacon chain for the validator's pubkey. To check for pending deposits up to the `expectedDepositRoot`, check the emitted logs from the beacon chain deposit contract.

Resolution

The Kelp team has acknowledged this issue and will use `stake32EthValidated()` and offchain checks to prevent front-running attacks as recommended above. The offchain checks were out of scope for this review and were not verified by Sigma Prime.

KLP2-02	Lost Funds After EigenLayer Operator-Initiated Undelegations		
Asset	`NodeDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

If the EigenLayer operator that the `NodeDelegator` is delegated to undelegates the `NodeDelegator` through `DelegationManager`, the total ETH in protocol will be misrepresented and the queued withdrawal will be stuck.

EigenLayer's `DelegationManager::undelegate()` allows operators to undelegate a staker that is delegated to them.

Since this undelegation is not executed through `NodeDelegator::undelegate()`, it bypasses Kelp's internal accounting in `NodeDelegator::undelegate()` and fails to call `LRTUnstakingVault::addSharesUnstaking()` as shown below:

```
function undelegate() external whenNotPaused onlyLRTManager {
    // ...
    for (uint256 i = 0; i < strategies.length;) {
        if (beaconChainETHStrategy == address(strategies[i])) {
            lrtUnstakingVault.addSharesUnstaking(LRTConstants.ETH_TOKEN, shares[i]);
        } else {
            address token = address(strategies[i].underlyingToken());
            lrtUnstakingVault.addSharesUnstaking(token, shares[i]);
        }
        // ...
    }
    // ...
}
```

As `sharesUnstaking` is not incremented after the undelegation, there are two impacts:

1. The queued withdrawal cannot be completed due to a revert caused by the underflow of `sharesUnstaking` when calling `lrtUnstakingVault.reduceSharesUnstaking()` - withdrawing funds cannot be recovered
2. The strategy shares in the queued withdrawal do not count to the protocol's total ETH - the `rsETH` price can be manipulated

Recommendations

To recover the stuck withdrawing funds, create a function in `LRTUnstakingVault` that allows an LRT operator to increase `sharesUnstaking` by registering queued withdrawals. To prevent the double counting of shares, ensure that withdrawals can only only be registered once and that `NodeDelegator::initiateUnstaking()` also registers queued withdrawals.

To prevent the manipulation of the `rsETH` price, create a circuit breaker that prevents the `rsETH` price from being updated when an operator has undelegated a staker by checking `DelegationManager::delegatedTo()` against `NodeDelegator::elOperatorDelegatedTo()`. Ensure that when the queued withdrawal is registered to `LRTUnstakingVault`,

the `elOperatorDelegatedTo` is reset to zero so that the `rsETH` price can be updated after the price manipulation risk has been mitigated.

Resolution

Queued withdrawals are now tracked in `LRTUnstakingVault` and a circuit breaker has been added to pause price updates, deposits, and rsETH withdrawals when there are untracked EigenLayer withdrawals.

This issue has been resolved in commit 38c2f47.

KLP2-03	Frozen Withdrawal Due To Wrong ETH Address In `completeUnstaking()`		
Asset	`NodeDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

If an asset is incorrectly set to match the `beaconChainETHStrategy` in the `completeUnstaking()` function, this asset cannot be withdrawn afterwards.

EigenLayer's `DelegationManager` contract ignores the inputted asset address when completing a withdrawal for `beaconChainETHStrategy`, allowing any asset address to be set in its place. If an LRT operator accidentally or maliciously sets the asset address to another asset with a pending withdrawal in `completeUnstaking()`, that asset will have its `sharesUnstaking` mapping reduced in `LRTUnstakingVault`:

```
for (uint256 i = 0; i < assetCount;) {
    lrtUnstakingVault.reduceSharesUnstaking(address(assets[i]), withdrawal.shares[i]);
    if (receiveAsTokens) {
        if (address(beaconChainETHStrategy) != address(withdrawal.strategies[i])) {
            balancesBefore[i] = assets[i].balanceOf(address(this));
        } else {
            balancesBefore[i] = address(this).balance;
        }
    }
    unchecked {
        i++;
    }
}
```

This will cause `sharesUnstaking` to revert due to an underflow when trying to complete the withdrawal for that asset, causing the withdrawal to be stuck.

Recommendations

In `completeUnstaking()`, reduce shares in `LRTConstants.ETH_TOKEN` for `beaconChainETHStrategy` instead of using the inputted `assets` array.

```
for (uint256 i = 0; i < assetCount;) {
    if (address(beaconChainETHStrategy) != address(withdrawal.strategies[i])) {
        lrtUnstakingVault.reduceSharesUnstaking(LRTConstants.ETH_TOKEN, withdrawal.shares[i]);
    } else {
        lrtUnstakingVault.reduceSharesUnstaking(address(assets[i]), withdrawal.shares[i]);
    }
    // ...
}
```

Resolution

The Kelp team has addressed the issue as recommended above in commit 1ad2f28.

KLP2-04	Incorrect `getETHEigenPodBalance()` Calculation Can Overinflate TVL		
Asset	`NodeDelegator.sol, LRTDepositPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `getETHEigenPodBalance()` function can incorrectly return 0 when there is a pod shares deficit in EigenLayer's `EigenPodManager` contract, leading to the protocol's calculated TVL being overinflated, as shown below:

```
/// @dev Returns the amount of eth staked in eigenlayer through this ndc
function getETHEigenPodBalance() external view override returns (uint256 ethStaked) {
   IEigenPodManager eigenPodManager = IEigenPodManager(lrtConfig.getContract(LRTConstants.EIGEN_POD_MANAGER));
   int256 nativeEthShares = eigenPodManager.podOwnerShares(address(this));

   if (nativeEthShares < 0) {
       // native eth shares are negative due to slashing and queue of more amount of eth withdrawal
       uint256 nativeEthSharesDeficit = uint256(-nativeEthShares);
       if (nativeEthSharesDeficit > stakedButUnverifiedNativeETH) {
           // @audit This incorrectly returns 0
           return 0;
       } else {
           return stakedButUnverifiedNativeETH - nativeEthSharesDeficit;
       }
   }

   return stakedButUnverifiedNativeETH + uint256(nativeEthShares);
}
```

This means that it is possible for a `NodeDelegator` to portray an overinflated balance in `LRTDepositPool::getTotalAssetDeposits()` if it has a queued withdrawal and has been slashed.

Consider the scenario where there is 1 `NodeDelegator` with 1 validator verified on `EigenPod` and no unverified validators:

```
initial state:           podOwnerShares =  32, stakedButUnverifiedNativeETH =  0, queuedShares =  0

queue withdrawal
for 32 shares:           podOwnerShares =  0 , stakedButUnverifiedNativeETH =  0, queuedShares =  32

validator gets slashed
for 16 ETH:              podOwnerShares = -16, stakedButUnverifiedNativeETH =  0, queuedShares =  32
```

In the scenario above, `LRTDepositPool::getTotalAssetDeposits()` for ETH will return 32 ETH as the pod shares deficit is ignored when `nativeEthSharesDeficit > stakedButUnverifiedNativeETH`.

Recommendations

Account for the pod shares deficit by returning an `int256` in `getETHEigenPodBalance()`. The potential negative value can be handled further up the call stack in `LRTDepositPool::getTotalAssetDeposits()` and

`LRTDepositPool::getETHDistributionData()`.

Resolution

The Kelp team has addressed the issue as recommended above in commit 1c2e477.

KLP2-05	`unlockQueue()` is Susceptible To Asset Price Manipulation		
Asset	`LRTWithdrawalManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The function `unlockQueue()` only checks for minimum prices and hence is vulnerable to price manipulation.

The `unlockQueue()` function enforces a minimum `rsETH` and asset price as follows:

```
if (rsETHPrice < minimumRsEthPrice) revert RsETHPriceMustBeGreaterMinimum(rsETHPrice);
if (assetPrice < minimumAssetPrice) revert AssetPriceMustBeGreaterMinimum(assetPrice);
```

The minimum prices are provided by the LRT operator when calling `unlockQueue()`, and are used to prevent withdrawals from being processed with manipulated prices.

However, this does not fully protect against price manipulation. If the `assetPrice` is manipulated to be a lot higher than intended, then withdrawals will end up with a significantly smaller payout, since `_calculatePayoutAmount()` takes the minimum of `request.expectedAssetAmount` and `newAssetAmount`:

```
function _calculatePayoutAmount(
    WithdrawalRequest storage request,
    uint256 rsETHPrice,
    uint256 assetPrice
)
    private
    view
    returns (uint256)
{
    uint256 currentReturn = (request.rsETHUnstaked * rsETHPrice) / assetPrice;
    return (request.expectedAssetAmount < currentReturn) ? request.expectedAssetAmount : currentReturn;
}
```

Recommendations

Instead of checking for minimum prices, the `unlockQueue()` function should check for expected prices for the `rsETH` and asset. The function should revert if the actual prices fall out of a set range from the expected prices.

Resolution

Maximum asset and `rsETH` price limits have been added to prevent price manipulation in the positive direction.

This issue has been resolved in commit 5b64c4d.

KLP2-06	`FeeReceiver` Balance Is Not Included In TVL		
Asset	`LRTDepositPool.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `getETHDistributionData()` function does not count the `FeeReceiver` contract's ETH balance, resulting in the `rsETH` price being smaller than intended.

Any execution layer rewards that are in the `FeeReceiver` contract are not included in the protocol's total ETH balance until the funds are sent to the `LRTDepositPool` via the `FeeReceiver::sendFunds()` function.

This creates a delay in the total ETH balance and, subsequently, the `rsETH` price. This will result in a pricing inefficiency that can be exploited by MEV searchers who can deposit into `LRTDepositPool` for a discount, and withdraw after the rewards have been accounted to the `rsETH` price.

This issue has a low impact as the `FeeReceiver` ETH balance only represents a small fraction of the total ETH in the protocol. Furthermore, MEV searchers are deterred from performing the arbitrage as they incur extra risk by waiting for the withdrawal delay in `LRTWithdrawalManager`. However, depending on market conditions and liquidity, the MEV searchers may perform the arbitrage on secondary markets, resulting in negative sell pressure on `rsETH`.

Recommendations

Include the `FeeReceiver` ETH balance in `getETHDistributionData()` so that the protocol's total ETH balance is more accurate, resulting in precise `rsETH` price.

Resolution

The Kelp team has acknowledged the issue with the following comment:

> *We currently use offchain automation to send the funds from the `FeeReceiver` to the `LRTDepositPool`. We will implement the recommended fix in the next upgrade.*

KLP2-07	Incorrect `stakedButUnverifiedNativeETH` Accounting Can Overinflate TVL		
Asset	`NodeDelegator.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

There are multiple possible attack paths involving `stakedButUnverifiedNativeETH` that cause staked ETH in validators to be double counted, overinflating the total ETH in the protocol and `rsETH` exchange rate. These attack paths are listed below:

1. Staking 32 ETH into a validator more than once.
2. Staking ETH into a validator and exiting the validator from the beacon chain without verifying withdrawal credentials on EigenPod.

In `stake32Eth()`, there are no checks to ensure that validators are only staked to once. Since `verifyWithdrawalCredentials()` only decreases `stakedButUnverifiedNativeETH` by 32 ETH and can only be called once per validator, it will not be correctly decremented back to 0, resulting in staked ETH from subsequent `stake32Eth()` calls being double counted.

The `incrementExtraStakeToReceive()` and `_reduceExtraStakes()` functions do not prevent the staked ETH from being double-counted, as they only decrement `stakedButUnverifiedNativeETH` once `NodeDelegator` receives the ETH.

EigenPod checkpoints track ETH balance changes, hence, it is possible for an unverified validator's withdrawn balance to be tracked by EigenPod and included as `podOwnerShares`. This scenario also causes the ETH to be double-counted, as `stakedButUnverifiedNativeETH` is not decremented as `podOwnerShares` increases.

Recommendations

To prevent the first scenario, create a registry of staked validator public keys and only allow validators to be staked into once. Keep in mind that already-existing validators will need to be added to this registry retroactively, which will incur significant gas cost.

To prevent the second scenario, replace the `incrementExtraStakeToReceive()` and `_reduceExtraStakes()` functions with an `emergencyReduceUnverifiedStake()` function that allows an LRT operator to instantly decrement `stakedButUnverifiedNativeETH`. Keep in mind that this is not a perfect fix, as it only provides a recovery measure instead of a preventative one.

Resolution

The first attack path has been mitigated by creating a registry of staked validator public keys as recommended above in commit d8854a3.

The Kelp team has acknowledged the second attack path with the following comment:

We have decided to not implement an emergency function as it introduces extra risks. To prevent this scenario, we will not exit validators without verification.

KLP2-08	`incrementExtraStakeToReceive()` Can Block Verifying Withdrawal Credentials		
Asset	`NodeDelegator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Incorrectly calling `incrementExtraStakeToReceive()` can prevent verifying validator withdrawal credentials in `EigenPod` due to the potential underflow of `stakedButUnverifiedNativeETH`.

The function `incrementExtraStakeToReceive()` is used to circumvent a double count issue that arises if an LRT operator mistakenly stakes to the same `pubkey` twice. This function increments the value of `extraStakeToReceive` and if the `NodeDelegator` contract receives ETH, the function `_reduceExtraStakes()` is called to reduce the extra double-counted stake:

```
function _reduceExtraStakes(uint256 extraStakeReceived) internal {
     if (extraStakeReceived <= 0) return;

     extraStakeToReceive -= extraStakeReceived;
     stakedButUnverifiedNativeETH -= extraStakeReceived;
     emit ExtraStakeReceived(extraStakeReceived);
 }
```

However, if an LRT operator accidentally or maliciously calls `incrementExtraStakeToReceive()` when there is no double count issue, that will prevent an unverified validator from verifying its withdrawal credentials on `EigenPod`, since `stakedButUnverifiedNativeETH` will underflow. This is because, when receiving rewards, `stakedButUnverifiedNativeETH` will be incorrectly decremented and hence, the call to the function `verifyWithdrawalCredentials()` will revert due to an underflow that will occur on line [**219**]:

```
function verifyWithdrawalCredentials(
    uint64 beaconTimestamp,
    BeaconChainProofs.StateRootProof calldata stateRootProof,
    uint40[] calldata validatorIndices,
    bytes[] calldata validatorFieldsProofs,
    bytes32[][] calldata validatorFields
)
    external
    onlyLRTOperator
{
    // reduce the eth amount that is verified
    stakedButUnverifiedNativeETH -= (validatorFields.length * (32 ether));
    // ...
}
```

Recommendations

Remove the `incrementExtraStakeToReceive()` function.

To prevent double-counting issues, create a registry that keeps track of every staked validator and ensure that in the `stake32Eth()` function the same pubkey cannot be staked twice.

Keep in mind that existing validators will also need to be added to this registry retroactively.

Resolution

This issue has been resolved as recommended above in commit d8854a3.

KLP2-09	`unlockQueue()` Potentially Uses Outdated `rsETH` Price		
Asset	`LRTWithdrawalManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `unlockQueue()` function does not update the `rsETH` price. Hence, users' withdrawals may be calculated based on an outdated `rsETH` price.

In `_calculatePayoutAmount()`, the `rsETH` price is used to calculate the payout amount of a user's withdrawal request.

```
function _calculatePayoutAmount(
        WithdrawalRequest storage request,
        uint256 rsETHPrice,
        uint256 assetPrice
    )
        private
        view
        returns (uint256)
    {
        uint256 currentReturn = (request.rsETHUnstaked * rsETHPrice) / assetPrice;
        return (request.expectedAssetAmount < currentReturn) ? request.expectedAssetAmount : currentReturn;
    }
```

`unlockQueue()` obtains the `rsETH` price by calling `LRTOracle::rsETHPrice()`, which may be outdated. If the price is outdated, the payout amount of a user's withdrawal request can be lower than intended, depending on the `rsETH` price when the request was made.

Recommendations

Update the `rsETH` price by calling `LRTOracle::updateRSETHPrice()` before calling `LRTOracle::rsETHPrice()` in `unlockQueue()`.

Resolution

The Kelp team has acknowledged this issue with the following comment:

> *To save on gas costs, we will manually update the price before calling `unlockQueue()`.*

KLP2-10	Lack Of Precision In `pricePercentageLimit`
Asset	`LRTOracle.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `pricePercentageLimit` variable uses 0 decimals of precision and hence, only integer percentage limits are allowed.

This is seen in `_isNewPriceOffLimit()` where `100` is used as the base for the percentage calculation:

```
function _isNewPriceOffLimit(uint256 oldPrice, uint256 newPrice) private view returns (bool) {
    // if oldPrice == newPrice, then no need to check
    if (oldPrice == newPrice) return false;
    // if pricePercentageLimit is 0, then no need to check
    if (pricePercentageLimit == 0) return false;

    // calculate the difference between old and new price
    uint256 diff = (oldPrice > newPrice) ? oldPrice - newPrice : newPrice - oldPrice;
    uint256 percentage = (diff * 100) / oldPrice;
    return percentage > pricePercentageLimit;
}
```

Recommendations

Use 2 decimals of precision for the `pricePercentageLimit` variable (i.e. `10_000` = 100%).

Resolution

The Kelp team has acknowledged this issue and will implement the recommended change in a future upgrade.

KLP2-11	`LRTOracle` Does Not Check Decimals
Asset	`LRTOracle.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `updatePriceOracleFor()` function does not check that the oracle uses 18 decimals before adding it.

The `_getTotalEthInProtocol()` function assumes that all price oracles use 18 decimals as it does not perform any decimal scaling. If any price oracle does not use 18 decimals, the result of `_getTotalEthInProtocol()` will be miscalculated.

Recommendations

Check that the oracle uses 18 decimals before adding it in `updatePriceOracleFor()`.

Resolution

The Kelp team has added an `updatePriceOracleForValidated()` function that performs a sanity check on the oracle's price before it is added in `LRTOracle`.

This issue has been resolved in commit 5808dff.

KLP2-12	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **MulDivs Are Missing Round Brackets**
   Related Asset(s): LRTOracle.sol
   The mulDiv in `updateRSETHPrice()` does not use round brackets.

```
uint256 rewardInETH = increaseInRsEthPrice * rsethSupply / 1e18;
uint256 rewardInRsETH = rewardInETH * 1e18 / tempRsETHPrice;
```

   Use round brackets to increase the readability of the code and minimise chances for errors.

```
uint256 rewardInETH = (increaseInRsEthPrice * rsethSupply) / 1e18;
uint256 rewardInRsETH = (rewardInETH * 1e18) / tempRsETHPrice;
```

2. **Incorrect Natspec Comments**
   Related Asset(s): /*
   There are multiple instances in the codebase where Natspec comments are outdated or incorrect:
   (a) `NodeDelegator::sendETHFromUnstakingVaultToNDC()`: Should be *"LRT unstaking vault"* instead of *"LRT deposit pool"*:
   (b) There are multiple functions where the Natspec specifies the incorrect access control (should be LRT operator instead of LRT manager):
   - `LRTConverter`: `swapEthToAsset()`, `transferAssetFromDepositPool()`
   - `LRTDepositPool`: `transferAssetToNodeDelegator()`, `transferETHToNodeDelegator()`, `transferAssetToLRTUnstakingVault()`, `transferETHToLRTUnstakingVault()`
   - `LRTUnstakingVault`: `transferAssetToNodeDelegator()`, `transferETHToNodeDelegator()`

   Edit the Natspec comments to fix the incorrect access control and update the descriptions to be more accurate.

3. **Protocol Fee Rounding Error**
   Related Asset(s): LRTOracle.sol
   There are unnecessary intermediate divisions in `updateRSETHPrice()` that may lead to rounding errors when calculating `rsethAmountToMintForProtocol`.

```
uint256 increaseInRsEthPrice = tempRsETHPrice - oldRsETHPrice; // new_price - old_price
uint256 rewardInETH = increaseInRsEthPrice * rsethSupply / 1e18;
uint256 rewardInRsETH = rewardInETH * 1e18 / tempRsETHPrice;
rsethAmountToMintForProtocol = lrtConfig.protocolFeeInBPS() * rewardInRsETH / 10_000;
```

   Calculate `rewardInRsETH` with one division as shown below:

```
uint256 increaseInRsEthPrice = tempRsETHPrice - oldRsETHPrice;
uint256 rewardInRsETH = increaseInRsEthPrice * rsethSupply / tempRsETHPrice
```

4. **Missing Sanity Checks**
   Related Asset(s): LRTConfig.sol & LRTOracle.sol

   - `LRTConfig::setProtocolFeeBps()` does not check if `_protocolFeeInBPS <= 10_000`.
   - `LRTConfig::updateAssetStrategy()` does not check the strategy's underlying token.
   - `LRTOracle::setPricePercentageLimit()` does not check if `_pricePercentageLimit <= 100`.

   Add these checks to ensure that the values are correct.

Recommendations

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

Resolution

The Kelp team has addressed the first two issues above in commit 2dc095d.

The third issue has been addressed by reworking the fee-calculation logic in commit 93b081b.

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 2 tests for test/tests-fork/LRTDepositPool.fork.t.sol:LRTDepositPoolForkTest
[SKIP] test_getETHDistributionData_FeeReceiverBalanceNotCounted2_Vuln() (gas: 0)
[SKIP] test_getETHDistributionData_FeeReceiverBalanceNotCounted_Vuln() (gas: 0)
Suite result: ok. 0 passed; 0 failed; 2 skipped; finished in 62.88ms (145.38µs CPU time)

Ran 2 tests for test/tests-fork/LRTUnstakingVault.fork.t.sol:LRTUnstakingVaultForkTest
[PASS] test_registerPendingWithdrawals_RemoveDelegatedTo() (gas: 1046685)
[PASS] test_registerPendingWithdrawals_RevertIf_WithdrawalAlreadyRegistered() (gas: 2461636)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 85.83ms (11.77ms CPU time)

Ran 3 tests for test/tests-fork/LRTWithdrawalManager.fork.t.sol:LRTWithdrawalManagerForkTest
[PASS] test_FullWithdrawalFlow() (gas: 2310488)
[PASS] test_unlockQueue_FrontRunningPriceManipulation_Vuln() (gas: 2849979)
[SKIP] test_unlockQueue_OutdatedRsETHPrice_Vuln() (gas: 0)
Suite result: ok. 2 passed; 0 failed; 1 skipped; finished in 85.72ms (51.19ms CPU time)

Ran 9 tests for test/tests-fork/NodeDelegator.fork.t.sol:NodeDelegatorForkTest
[PASS] test_ForcedUndelegation_FrozenQueuedWithdrawal_Vuln() (gas: 1229439)
[PASS] test_ForcedUndelegation_RsETHPriceManipulation_Vuln() (gas: 6422185)
[PASS] test_completeUnstaking_BeaconChainETHStrategyWrongAsset_Vuln() (gas: 4284280)
[PASS] test_completeUnstaking_RevertIf_UnaccountedWithdrawals() (gas: 1457426)
[PASS] test_getETHEigenPodBalance_OverInflateTVL_Vuln() (gas: 5077049)
[PASS] test_incrementExtraStakeToReceive_InsufficientDoubleCountPrevention_Vuln() (gas: 2017224)
[PASS] test_reduceExtraStakes_VerifyWithdrawalCredentialsDoS_Vuln() (gas: 1946057)
[SKIP] test_stakedButUnverifiedNativeETH_DoubleCounting_Vuln() (gas: 0)
[PASS] test_verifyWithdrawalCredentials() (gas: 5417350)
Suite result: ok. 8 passed; 0 failed; 1 skipped; finished in 231.12ms (372.45ms CPU time)

Ran 5 tests for test/tests-fork/LRTOracle.fork.t.sol:LRTOracleForkTest
[PASS] testDifferentialFuzz1_updateRSETHPrice_RoundingError(uint256,uint256,uint256) (runs: 1005, µ: 5099394, ~: 5099546)
[PASS] testDifferentialFuzz2_updateRSETHPrice_RoundingError(uint256,uint256,uint256) (runs: 1005, µ: 5107812, ~: 5107952)
[PASS] testFuzz_updateRSETHPrice_FeeCalculation(uint256,uint256) (runs: 1005, µ: 6916927, ~: 6917013)
[PASS] testFuzz_updateRSETHPrice_FeeRoundingError() (gas: 77863818)
[PASS] testFuzz_updateRSETHPrice_RoundingErrorToZero(uint256,uint256,uint256) (runs: 1005, µ: 2152520, ~: 2152666)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 55.77s (135.67s CPU time)

Ran 5 test suites in 55.78s (56.24s CPU time): 17 tests passed, 0 failed, 4 skipped (21 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.

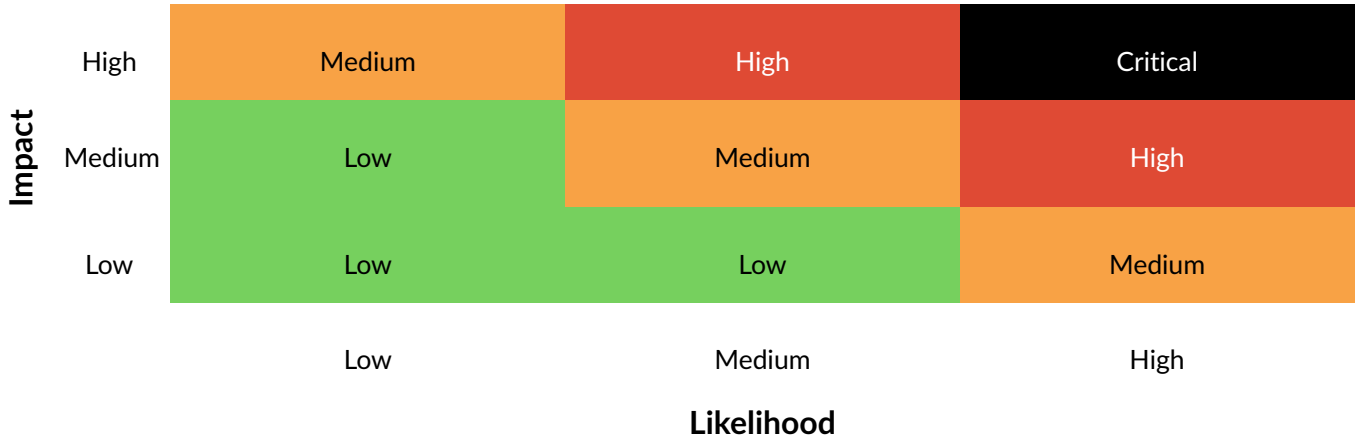

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].