Kelp

June Upgrades

Security Assessment Report

Version: 2.1

June, 2025

Contents

Introduction

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

The review focused solely on the security aspects of the Solidity implementation of the contracts, 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 components in scope. 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 components 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*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Kelp components in scope.

Overview

Kelp DAO is a liquid restaking protocol, building on top of EigenLayer. It gives users access to multiple benefits such as staking and restaking rewards, DeFi and liquidity.

This review focuses on upgrades introduced in June 2025 at commit `0c7eea5`.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Kelp-DAO repository.

The scope of this time-boxed review was strictly limited to the following files at the commit diff d4f41e0..0c7eea5:

- FeeReceiver.sol
- L1Vault.sol
- L1VaultV2.sol
- LRTConfig.sol
- LRTConverter.sol
- LRTDepositPool.sol
- LRTOracle.sol
- LRTUnstakingVault.sol
- LRTWithdrawalManager.sol
- NodeDelegator.sol
- NodeDelegatorHelper.sol
- L2/RETHTokenWrapper.sol
- RSETH.sol
- utils/HashStorage.sol
- utils/WadMath.sol
- KERNEL/KernelTop100MerkleDistributor.sol
- utils/LRTConfigRoleChecker.sol
- utils/LRTConstants.sol
- utils/UtilLib.sol

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

The fixes for the identified issues were assessed at PR-245.

Approach

The security assessment covered components written in Solidity.

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 the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

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 6 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Low: 3 issues.
- Informational: 2 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Kelp components 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 components, including 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 components(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
KLP8-01	Stale `RSETH` Price Referenced In Core Operations	High	Closed
KLP8-02	Missing Role Assignments For `LRTOracle`	Low	Closed
KLP8-03	`RSETH` Minting Allowed During Paused State	Low	Resolved
KLP8-04	Unrestricted Modification Of `vestingStartTimestamp` May Lead To Arithmetic Underflow	Low	Resolved
KLP8-05	Misleading Price Event Emission `RsETHPriceDecrease`	Informational	Resolved
KLP8-06	Redundant Timestamp Initial Setting In `verifyAndUpdateClaim()`	Informational	Resolved

KLP8-01	Stale RSETH Price Referenced In Core Operations		
Asset	LRTDepositPool.sol, LRTWithdrawalManager.sol		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The deposit and withdrawal logic fails to ensure that RSETH pricing is up-to-date at the time of operation. This introduces a temporal misalignment between the total value locked (TVL) and the token price used in minting or redemption calculations, resulting in valuation inaccuracies and potential unfair outcomes for protocol participants.

The following functions are affected:

- `LRTDepositPool.depositETH()`
- `LRTDepositPool.depositAsset()`
- `LRTWithdrawalManager.initiateWithdrawal()`
- `LRTWithdrawalManager.instantWithdrawal()`

A typical exploitation scenario involves the protocol receiving additional ETH (e.g. from staking rewards or unsolicited transfers), increasing the actual TVL. However, unless the RSETH price is manually updated, it remains stale. If users deposit or withdraw during this window, they may receive incorrect RSETH quantities or asset allocations:

- Depositors might mint more RSETH than is fair based on outdated pricing.
- Withdrawers might redeem RSETH for more assets than appropriate.
- Arbitrageurs may exploit these windows for financial gain.

These inconsistencies compromise the integrity of the minting and redemption mechanisms and could lead to loss of value for honest participants or misrepresentation of circulating supply.

This issue is rated as medium impact because it does not result in direct unauthorised asset transfers or control over protocol state. However, it is of high likelihood, as price updates are manual and subject to delays, creating frequent opportunities for exploitation in normal operation or during reward accrual periods.

Recommendations

Perform price updates by calling `LRTOracle.updateRSETHPrice()` before initiating any deposits or withdrawals.

Resolution

The finding has been closed with the following comment from the development team:

"It is a design choice. If we allow a user to inadvertently deposit into the protocol and at the same time call the update price function, it will cause more harm. The user could have their transaction cost (increase) from a 2 dollar to a 30 dollar transaction. It would be detrimental to small users. We are confident that the architecture choice we made is best for the protocol."

KLP8-02	Missing Role Assignments For `LRTOracle`		
Asset	`LRTOracle.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The deployed configuration fails to assign the required privileges to the `LRTOracle` contract, preventing it from executing critical operations such as pausing the system during emergencies and minting protocol fees. This misconfiguration can lead to silent failures, undermining safety mechanisms and core protocol behaviour.

Within `updateRSETHPrice()`, the oracle is designed to safeguard the system by pausing deposits and withdrawals when a significant price drop is detected:

```
if (isPriceDecreaseOffLimit) {
    IPausable(lrtConfig.getContract(LRTConstants.LRT_DEPOSIT_POOL)).pause();
    IPausable(lrtConfig.getContract(LRTConstants.LRT_WITHDRAW_MANAGER)).pause();
    return;
}
```

However, these `pause()` calls will revert at runtime because the `LRTOracle` contract lacks the `PAUSER_ROLE` required to perform them. As a result, the intended emergency mechanism becomes non-functional, potentially leaving the protocol vulnerable during severe market volatility.

Additionally, the same function attempts to mint protocol fees in the form of `RSETH` when new `ETH` is detected in the system:

```
if (rsethAmountToMintAsProtocolFee > 0) {
    IRSETH(rsETHTokenAddress).mint(
        lrtConfig.getContract(LRTConstants.PROTOCOL_TREASURY),
        rsethAmountToMintAsProtocolFee
    );
}
```

This minting operation requires the `MINTER_ROLE`, which has also not been granted to the oracle. Without it, protocol fees will not be distributed as intended, silently breaking a key economic function and potentially affecting treasury funding.

Recommendations

Ensure that the `LRTOracle` contract is granted the following roles during deployment or initialisation:

1. `PAUSER_ROLE` on both the `LRTDepositPool` and `LRTWithdrawalManager` contracts, to enable emergency halts during price shocks.
2. `MINTER_ROLE` on the `RSETH` token contract, to allow correct distribution of protocol fees.

Resolution

The development team have acknowledged the issue and determined that no fix is needed as all roles are assigned correctly in a production setting on the ETH mainnet.

KLP8-03	RSETH Minting Allowed During Paused State		
Asset	LRTOracle.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The system exhibits inconsistent behaviour during emergency scenarios triggered by sharp price declines. While core protocol operations are correctly paused to prevent potentially unsafe user interactions, fee minting logic remains active, allowing the protocol to continue issuing new tokens even while in a paused state.

When the `updateRSETHPrice()` function detects a critical drop in price, it correctly triggers an emergency pause by invoking the `pause()` function on both the deposit and withdrawal contracts:

```
if (isPriceDecreaseOffLimit) {
    IPausable lrtDepositPool = IPausable(lrtConfig.getContract(LRTConstants.LRT_DEPOSIT_POOL));
    IPausable withdrawalManager = IPausable(lrtConfig.getContract(LRTConstants.LRT_WITHDRAW_MANAGER));
    lrtDepositPool.pause();
    withdrawalManager.pause();
    return;
}
```

However, if ETH is subsequently donated to the protocol (thereby increasing the total value locked), the same `updateRSETHPrice()` function continues executing logic that calculates a protocol fee based on the change in ETH balance:

```
if (totalETHInProtocol > previousTVL) {
    uint256 rewardAmount = totalETHInProtocol - previousTVL;
    protocolFeeInETH = (rewardAmount * lrtConfig.protocolFeeInBPS()) / 10_000;
}
```

It then mints new RSETH to the protocol treasury despite the system being in a paused state:

```
if (protocolFeeInETH > 0) {
    // Calculate rsETH amount to mint as protocol fee
    uint256 rsethAmountToMintAsProtocolFee = protocolFeeInETH.divWad(newRsETHPrice);

    // rest of code

    if (rsethAmountToMintAsProtocolFee > 0) {
        address treasury = lrtConfig.getContract(LRTConstants.PROTOCOL_TREASURY);
        IRSETH(rsETHTokenAddress).mint(treasury, rsethAmountToMintAsProtocolFee);
        emit FeeMinted(treasury, rsethAmountToMintAsProtocolFee);
    }
}
```

This behaviour is problematic because it undermines the emergency pause mechanism. The intention behind pausing is to halt all sensitive operations during volatile market conditions. Continuing to mint RSETH during such times may erode confidence in the pause mechanism, create accounting inconsistencies, or allow strategic manipulation of protocol parameters.

Recommendations

Incorporate explicit checks within `updateRSETHPrice()` to ensure that protocol fee minting is only performed when the system is fully active. If either the deposit pool or the withdrawal manager is paused, fee minting should be skipped to preserve operational consistency and uphold the intended effects of the emergency pause.

Resolution

The finding was addressed in PR-245.

KLP8-04	Unrestricted Modification Of `vestingStartTimestamp` May Lead To Arithmetic Underflow		
Asset	`KernelTop100MerkleDistributor.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The function `setVestingStartTimestamp()` allows the owner to update the `vestingStartTimestamp` at any time without verifying if vesting has already begun:

```
function setVestingStartTimestamp(uint256 _vestingStartTimestamp) external onlyOwner {
    if (_vestingStartTimestamp == 0) {
        revert ZeroValueProvided();
    }
    vestingStartTimestamp = _vestingStartTimestamp;
    emit VestingStartTimestampSet(vestingStartTimestamp);
}
```

If the vesting start timestamp is changed after some vesting has already occurred, previously claimed amounts will no longer align with the updated vesting schedule. For example:

1. Vesting begins on Day 3.
2. A user successfully claims tokens on Day 5.
3. On Day 6, the owner resets `vestingStartTimestamp` to Day 6.
4. When the same user attempts to claim again on Day 7, the system calculates the vested amount based only on the interval from Day 6 to Day 7. However, the user's previous claim was based on the original Day 3 to Day 5 vesting window.

   This mismatch can cause the calculation of `unclaimedAmount` to underflow:

   ```
   uint256 unclaimedAmount = totalVestedAmount - userClaim.amountClaimed;
   ```

   In this case, `totalVestedAmount` is recalculated from the new start time (Day 6), while `userClaim.amountClaimed` still reflects the earlier vesting (Day 3 to Day 5), resulting in an arithmetic underflow and transaction revert.

Recommendations

Update the implementation of `setVestingStartTimestamp()` to include the following safeguards:

1. Reject any modification if vesting has already commenced, by ensuring the current block timestamp is less than the existing `vestingStartTimestamp`.
2. Ensure that the new vesting start time is not in the past relative to the current block timestamp, to avoid retroactive schedule manipulation.

Resolution

The finding was addressed in PR-245.

KLP8-05	Misleading Price Event Emission `RsETHPriceDecrease`
Asset	`LRTOracle.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `updateRSETHPrice()` function emits a `RsETHPriceDecrease` event whenever the new price is below the historical highest price, regardless of whether the price has actually decreased from its previous value:

```
if (newRsETHPrice < highestRsethPrice) {
    uint256 diff = highestRsethPrice - newRsETHPrice;
    // normalizing to 1e18
    bool isPriceDecreaseOffLimit =
        pricePercentageLimit > 0 && diff > pricePercentageLimit.mulWad(highestRsethPrice);

    emit RsETHPriceDecrease(highestRsethPrice, newRsETHPrice);

    // rest of code
}
```

The event name suggests an absolute price drop, but it even triggers in scenarios when the price has actually increased from its previous value but simply has not surpassed the all-time high. This naming convention could mislead monitoring systems or users interpreting the price trends

Recommendations

To improve accuracy and clarity, it is recommended to consider either:

1. Renaming the event to better reflect its actual purpose (e.g., `RsETHPriceBelowPeak`)
2. Implementing separate events to distinguish between:
    - Actual price decreases
    - Price increases that remain below historical highs

Resolution

The finding was addressed in PR-245.

KLP8-06	Redundant Timestamp Initial Setting In `verifyAndUpdateClaim()`
Asset	`KernelTop100MerkleDistributor.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In the `KERNEL` token claiming process, the function `verifyAndUpdateClaim()` includes logic that sets `lastClaimTimestamp` to `vestingStartTimestamp` if it is zero:

```
if (userClaim.lastClaimTimestamp == 0) {
    userClaim.lastClaimTimestamp = vestingStartTimestamp;
}
```

However, this setting is redundant because the function `_getUnclaimedVestedAmount()` already handles this edge case by defaulting to `vestingStartTimestamp` when `lastClaimTimestamp` is zero:

```
uint256 startTime = userClaim.lastClaimTimestamp > 0 ? userClaim.lastClaimTimestamp : vestingStartTimestamp;
```

Recommendations

To improve code efficiency and reduce redundancy, the initial timestamp check in `verifyAndUpdateClaim()` should be removed, as `_getUnclaimedVestedAmount()` already ensures the correct start time is used.

Resolution

The finding was addressed in PR-245.

Appendix A 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.

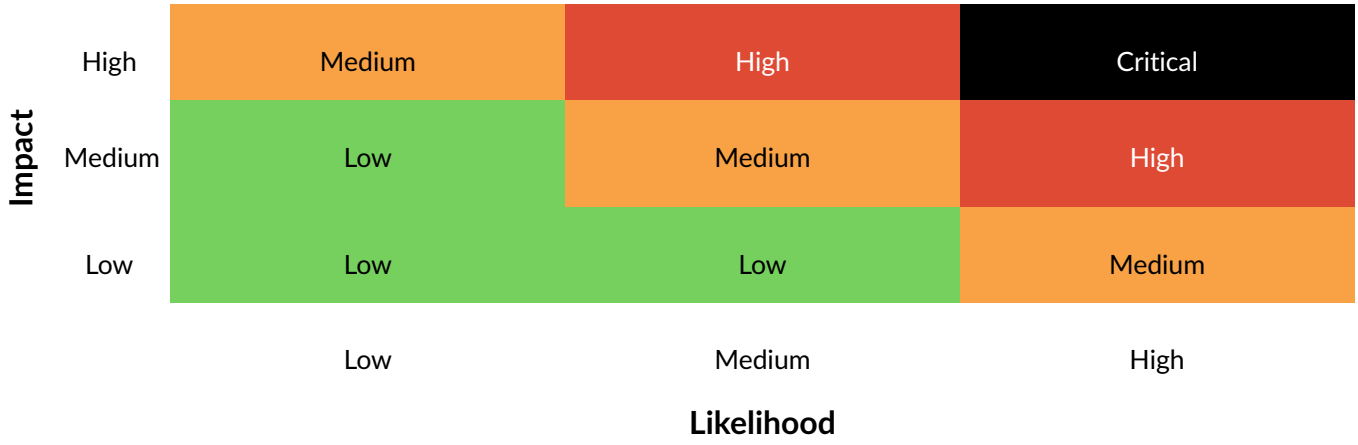

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].