AstraSec

Liquid Royalty Security Audit Report

December 11, 2025

Contents

1 Introduction

1.1 About Liquid Royalty

Liquid Royalty is a decentralized, blockchain-based financial system that connects traditional finance with the emerging DeFi ecosystem. It provides a framework for offering return-bearing, liquid, and transparent financial products backed by e-commerce revenue streams.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/stratosphere-network/LiquidRoyaltyContracts.git
- CommitID: b8234e5

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/stratosphere-network/LiquidRoyaltyContracts.git
- CommitID: 07b5f14

Please note that the scope of this audit is limited to the BaseVault.sol, JuniorVault.sol, ReserveVault.sol, and UnifiedSeniorVault.sol contracts themselves. This audit does not cover the security of contracts that interact with them and composability risks arising from such interactions.

1.3 Revision History

Version	Date	Description
v1.0	December 11, 2025	Initial Audit

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the Liquid Royalty protocol. Throughout this audit, we identified a total of 6 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	2	–	–	2
High	1	1	–	–
Medium	2	–	–	2
Low	1	–	–	1
Informational	–	–	–	–
Undetermined	–	–	–	–

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Acknowledged
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and re-mediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

3.3.1 [C-1] Unauthorized Withdrawal in BaseVault::_withdraw()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
BaseVault.sol	Business Logic	High	High	Addressed

The BaseVault::_withdraw() function overrides the ERC4626 withdrawal logic but omits any authorization check that links caller and owner. It allows withdraw()/redeem() to be executed with an arbitrary owner address, then burns that owner's shares and transfers netAssets to the specified receiver without verifying that the caller is the owner or has sufficient allowance. Consequently, any attacker can withdraw someone else's assets from the vault without their approval, leading directly to unauthorized fund theft.

LiquidRoyaltyContracts - BaseVault.sol

```
938 function _withdraw(
939     address caller,
940     address receiver,
941     address owner,
942     uint256 assets,
943     uint256 shares
944 ) internal virtual override {
945     // Calculate 1% withdrawal fee
946     uint256 withdrawalFee = (assets * MathLib.WITHDRAWAL_FEE) / MathLib.PRECISION;
947     uint256 netAssets = assets - withdrawalFee;
948
949     ...
950
951     // Burn shares from owner
952     _burn(owner, shares);
953
954     // Transfer net assets to receiver (after 1% fee)
955     _stablecoin.safeTransfer(receiver, netAssets);
956
957     // Transfer fee to treasury (if treasury is set)
958     if (_treasury != address(0) && withdrawalFee > 0) {
959         _stablecoin.safeTransfer(_treasury, withdrawalFee);
960         emit WithdrawalFeeCharged(owner, withdrawalFee, netAssets);
961     }
962
963     // Track capital outflow: decrease vault value by actual assets withdrawn (full amount including fee)
964     _vaultValue -= assets;
965
966     emit Withdraw(caller, receiver, owner, assets, shares);
967 }
```

Remediation Add necessary checks so only the owner or an approved spender can call _withdraw() for a given owner.

3.3.2 [C-2] Lack of Input Validation in depositLP()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
JuniorVault.sol UnifiedSeniorVault.sol	Business Logic	High	High	Addressed

JuniorVault::depositLP() and UnifiedSeniorVault::depositLP() accept any ERC20 token as lpToken and do not require it to match the island token configured in KodiakHook. When a deposit is later cancelled, rejected, or expired, the contracts unconditionally call kodiakHook.transferIslandLP(recipient, deposit.amount), which always sends genuine island token and ignores the originally supplied lpToken address. This mismatch allows an attacker to deposit arbitrary worthless token and then, via cancellation or expiry, withdraw the same amount of real island token from the vault.

LiquidRoyaltyContracts - JuniorVault.sol

```
237 function depositLP(address lpToken, uint256 amount) external returns (uint256 depositId) {
238     if (lpToken == address(0)) revert ZeroAddress();
239     if (amount == 0) revert InvalidAmount();
240     if (address(kodiakHook) == address(0)) revert KodiakHookNotSet();
241
242     // Transfer LP from user to hook
243     IERC20(lpToken).safeTransferFrom(msg.sender, address(kodiakHook), amount);
244
245     // Create pending deposit
246     depositId = _nextDepositId++;
247     uint256 expiresAt = block.timestamp + DEPOSIT_EXPIRY_TIME;
248
249     _pendingDeposits[depositId] = PendingLPDeposit({
250         depositor: msg.sender,
251         lpToken: lpToken,
252         amount: amount,
253         timestamp: block.timestamp,
254         expiresAt: expiresAt,
255         status: DepositStatus.PENDING
256     });
257
258     _userDepositIds[msg.sender].push(depositId);
259
260     emit PendingLPDepositCreated(depositId, msg.sender, lpToken, amount, expiresAt);
261 }
```

LiquidRoyaltyContracts - JuniorVault.sol

```
329 function cancelPendingDeposit(uint256 depositId) external {
330     PendingLPDeposit storage deposit = _pendingDeposits[depositId];
331
332     if (deposit.depositor == address(0)) revert DepositNotFound();
333     if (deposit.depositor != msg.sender) revert NotDepositor();
334     if (deposit.status != DepositStatus.PENDING) revert DepositNotPending();
335
336     // Transfer LP back from hook to depositor
337     kodiakHook.transferIslandLP(deposit.depositor, deposit.amount);
338
339     // Update status
340     deposit.status = DepositStatus.CANCELLED;
341
342     emit PendingLPDepositCancelled(depositId, deposit.depositor);
343 }
```

Remediation Add a validation in depositLP() to ensure that lpToken matches the island token configured in KodiakHook.

3.3.3 [H-1] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	High	Medium	Acknowledged

The Liquid Royalty protocol relies on multiple privileged accounts that possess extensive control over critical operations, introducing notable centralization risks. These accounts, assigned distinct roles, can unilaterally influence the protocol's functionality and integrity. Below are key examples of privileged functions and their associated roles:

- ➔ **Contract Updater Role:** Authorizes contract upgrades via _authorizeUpgrade(), allowing changes to core protocol logic (deposits, withdrawals, fees, accounting).
- ➔ **Admin Role:** Manages vault configuration (LP whitelists, hooks, treasury), executes rebase(), pauses/unpauses vaults, triggers emergencyWithdraw(), and can burn shares using adminBurn(). These operations can modify how user assets are invested, valued, or made available for withdrawal.
- ➔ **Liquidity Manager Role:** Deploys user funds into LP positions (investInLP), withdraws LP tokens, approves/rejects LP deposits, and mints management fees. Controls where user capital is invested and can affect returns.
- ➔ **Price Feed Manager Role:** Updates vault valuation through updateVaultValue() and setVaultValue(), which drive share pricing and profit/loss calculations. Changes made by this role influence how gains and losses are reflected across users.
- ➔ **Seeder Role:** Can mint or transfer shares on behalf of users, effectively reallocating value within the vault.

LiquidRoyaltyContracts - BaseVault.sol

```
887 function _authorizeUpgrade(address newImplementation) internal override onlyContractUpdater {
888     // Admin authorization check via modifier
889 }
```

LiquidRoyaltyContracts - BaseVault.sol

```
921 function adminBurn(address account, uint256 amount) public virtual onlyAdmin {
922     _burn(account, amount);
923 }
```

LiquidRoyaltyContracts - BaseVault.sol

```
785 function setVaultValue(uint256 newValue) public virtual onlyPriceFeedManager {
786     // Allow 0 to enable truly empty vault state for first deposit
787     uint256 oldValue = _vaultValue;
788     _vaultValue = newValue;
789     _lastUpdateTime = block.timestamp;
790
791     // Calculate BPS for event logging
792     int256 bps = 0;
793     if (oldValue > 0) {
794         bps = int256((newValue * 10000 / oldValue)) - 10000;
795     }
796
797     emit VaultValueUpdated(oldValue, _vaultValue, bps);
798
799     // Hook for derived contracts to execute post-update logic
800     _afterValueUpdate(oldValue, _vaultValue);
801 }
```

Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.
Response By Team This issue has been confirmed by the team.

3.3.4 [M-1] Reentrancy Risk Caused by Breaking CEI Pattern

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Business Logic	High	Low	Addressed

In BaseVault::_withdraw(), the function burns the user's shares (line 947) but defers updating _vaultValue (line 953) until after external _stablecoin.safeTransfer() calls to the receiver. If _stablecoin supports callbacks (e.g., ERC777 tokensReceived()) and either receiver is attacker-controlled, the attacker can reenter withdraw()/redeem() during the hook. During reentrancy, the vault's share price (_vaultValue / totalSupply()) remains overstated because _vaultValue hasn't been reduced while totalSupply() has already decreased from the burned shares. This allows the attacker to withdraw more underlying tokens than their remaining shares should entitle, with each reentrant iteration compounding the imbalance by burning additional shares while further delaying the _vaultValue update.

Remediation Refactor _withdraw() to follow the CEI (Checks-Effects-Interactions) pattern to prevent reentrancy attacks. Meanwhile, the following functions should also follow the CEI pattern:: UnifiedSeniorVault::withdraw(), JuniorVault/UnifiedSeniorVault::depositLP(), and JuniorVault/UnifiedSeniorVault::approveLPDeposit()/rejectLPDeposit()/cancelPendingDeposit()/claimExpiredDeposit().

3.3.5 [M-2] Potential Sandwich/MEV Attacks in Vault Contracts

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
BaseVault.sol UnifiedSeniorVault.sol	Business Logic	High	High	Addressed

While examining the implementation of BaseVault contract, we notice an attacker can sandwich the price feed manager's call to setVaultValue() or updateVaultValue() to extract value from other users. The attack sequence is:

- ➔ (1) **deposit():** deposit assets when _vaultValue is low, receiving shares at a favorable rate based on the current _vaultValue / totalSupply() ratio;
- ➔ (2) **setVaultValue()/updateVaultValue():** the price feed manager increases _vaultValue, raising the per-share value;
- ➔ (3) **withdraw():** immediately withdraw, converting shares back to assets at the higher price before other users can benefit.

Because the value update increases _vaultValue, the attacker captures a disproportionate share of the value increase, effectively extracting value that should be distributed across all shareholders. This is especially profitable if the attacker can frontrun the price update transaction or if the update is predictable.

Morever, the UnifiedSeniorVault::rebase() function shares the similar issue.

LiquidRoyaltyContracts - BaseVault.sol

```
785 function setVaultValue(uint256 newValue) public virtual onlyPriceFeedManager {
786     // Allow 0 to enable truly empty vault state for first deposit
787     uint256 oldValue = _vaultValue;
788     _vaultValue = newValue;
789     _lastUpdateTime = block.timestamp;
790
791     ...
792 }
```

Remediation Add a withdrawal delay mechanism to prevent potential sandwich attacks. At the same time, critical transactions should be routed through a privacy pool.

3.3.6 [L-1] Incompatibility with Non-Standard ERC20 Tokens

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Compatibility	Low	Low	Addressed

The BaseVault::investInLP() function transfers stablecoins to the whitelisted LP using _stablecoin.transfer(lp, amount). Since the underlying IERC20 interface declares transfer() as returning a bool, calling this function on non-standard tokens such as USDT (which does not return a value) will cause the transaction to revert. This prevents the vault from investing in the LP when USDT is used as the stablecoin and may permanently block protocol operations.

A similar issue exists in several other functions across both contracts. In BaseVault, the functions investInLP(), withdrawLPTokens(), and transferToSenior() exhibit the same fragility. Likewise, in UnifiedSeniorVault, the functions investInLP(), withdrawLPTokens(), deposit(), withdraw(), and emergencyWithdraw() are also affected.

LiquidRoyaltyContracts-master - BaseVault.sol

```
334 function investInLP(address lp, uint256 amount) external onlyLiquidityManager {
335     ...
336     // Check vault has sufficient stablecoin balance
337     uint256 vaultBalance = _stablecoin.balanceOf(address(this));
338     if (vaultBalance < amount) revert InsufficientBalance();
339
340     // Transfer stablecoins from vault to LP
341     _stablecoin.transfer(lp, amount);
342
343     emit LPInvestment(lp, amount);
344 }
```

Remediation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related transfer() and transferFrom().

.

4 Appendix

4.1 About AstraSec

AstraSec is a premier blockchain security firm dedicated to delivering high-quality auditing services for blockchain-based protocols. Composed of veteran security researchers with extensive experience across the DeFi landscape, our team maintains an unwavering commitment to excellence and precision. AstraSec's comprehensive methodology and deep understanding of blockchain architecture enable us to ensure protocol resilience, making us a trusted partner for our clients.

4.2 Disclaimer

The content of this audit report is for informational purposes only and does not constitute legal, financial, or investment advice. The findings, views, and conclusions presented herein are based on the code and documentation provided at the specific time of the audit and may be subject to risks and uncertainties not detected during the assessment.

While AstraSec exerts every effort to ensure the accuracy and completeness of this report, the information is provided on an "as is" basis. We assume no responsibility for any errors, omissions, or inaccuracies. Users should conduct their own independent due diligence and consult with professional advisors before making any investment or deployment decisions. AstraSec shall not be held liable for any direct or indirect losses, damages, or consequences arising from reliance on this report.

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://x.com/AstraSecAI