

# SuperRare – Liquid Edition Security Review

---

**Prepared by:** Ayeslick Audits

**Date:** November 25, 2025

## About

---

Ayeslick Audits is a boutique smart contract security firm with a strong record of identifying high-impact vulnerabilities across top-tier NFT and DeFi protocols since 2022.

## Disclaimer

---

While a smart contract security review is comprehensive, it cannot guarantee the total absence of vulnerabilities due to time, resource, and expertise constraints. In my review, I aim to uncover as many issues as possible, but this does not ensure complete security. To bolster your smart contract's security, subsequent reviews, participation in bug bounty programs, and ongoing on-chain monitoring are highly advisable.

## Introduction

---

A security review was conducted on SuperRare's Liquid Edition system, covering token launch mechanics, bonding-curve pricing, swap reward processing, and RARE burn logic.

## Review Commit Hash

cd6165bc3dd03a953c9e15cfc2915abfa01330f6

## Fixes Review Commit Hash

f730ecea6f1e4a845be0d3b725afa197caf5295

## Risk Classification

---

### Impact

- **CRITICAL** – Directly exploitable vulnerability leading to stolen/lost/locked funds or catastrophic denial of service.
- **HIGH** – Vulnerability impacting correct system function, incorrect states, or denial of service under certain conditions.
- **MEDIUM** – No direct asset risk, but protocol function, economics, or availability may be impacted.
- **LOW** – Best-practice violations, inefficiencies, or low-impact incorrect usage of primitives.

### Likelihood

- **HIGH** – Exploitation requires assumptions consistent with on-chain conditions.
- **MEDIUM** – Exploitable under incentivized but less common circumstances.
- **LOW** – Requires unrealistic assumptions or prohibitive cost.

## Scope

---

The following contracts were reviewed:

- `Liquid.sol`
- `LiquidFactory.sol`
- `RAREBurner.sol`

# Findings

---

## MEDIUM — LiquidFactory

### Issue:

`setRareBurnFeeBPS`, `setProtocolFeeBPS`, and `setReferrerFeeBps` each independently validate that all three fees sum to 100%. This makes updating the fee distribution impossible after deployment because changing any single fee violates the validation before the remaining fees can be adjusted.

### Recommendation:

Allow updating all fees at once (e.g., a single function setting all three values atomically).

### Note:

SuperRare identified and remediated this issue during the audit.

---

## MEDIUM — Liquid.sol

### Spot-Price Slippage Bypass in `_processLiquidRewards`

The `_processLiquidRewards` function uses a spot price quote (obtained inside the same transaction) to compute `minOut` for reward swaps.

An attacker can front-run `_handleSecondaryRewards()` to push the pool price downward, causing:

1. The contract to query a manipulated price.
2. The quote to be accepted as a valid `minOut`.
3. The swap to execute at the attacker-manipulated rate.
4. The attacker to back-run and profit.

Since the quote and swap occur in the same block, this “slippage protection” provides no actual protection.

### Recommendation:

Use a TWAP-based price or another manipulation-resistant oracle.

**Note:**

SuperRare decided to provide a `slippageBps` for harvesting. Caller pings `quoteHarvestParams` which they can then pass to `harvestSecondaryRewards`.

---

## MEDIUM — Liquid.sol

### Unrecoverable ETH on Partial Fills

When a swap hits `sqrtPriceLimitX96`, the Uniswap V4 swap may consume only part of the ETH sent.

However:

1. The callback `_unlockSwapBuy` settles only the used ETH with the pool.
2. The `buy` function incorrectly assumes all ETH was consumed.
3. No refund mechanism exists.
4. The unused ETH remains stuck in the Liquid contract forever.

**Recommendation:**

Implement refund logic to return unused ETH back to the user.

**Note:**

SuperRare made swapping an atomic all-or-nothing call.

---

## MEDIUM — Liquid.sol

### Fee Overcharge on Partial Fills

The contract deducts trading fees from the full `msg.value` before the swap occurs.

If a swap partially fills:

- Users pay fees on the entire attempted input,
- Not the actual ETH used,

- Resulting in fee overpayment.

**Recommendation:**

Calculate the fee from the actual `ethUsed` , not the initial `msg.value` .

**Note:**

SuperRare made swapping an atomic all-or-nothing call.

---

## MEDIUM — Liquid.sol

`minPayoutSize` validation occurs before fee deduction

`minPayoutSize` is validated against the raw swap output, before fees are subtracted.

Thus, final user payout can fall below the user-specified minimum.

**Recommendation:**

Validate the post-fee amount.

**Note:**

SuperRare decided to validate the slippage after the fee deduction.

---

## INFO — Liquid.sol

`tx.origin` check incompatible with EIP-7702

The `receive()` function uses:

```
tx.origin != msg.sender
```

EIP-7702 enables EOAs to delegate execution to contract code, causing `tx.origin == msg.sender` even for contract-executed transactions.

This breaks the intended “no contract callers” restriction.

**Note:**

SuperRare decided to remove `tx.origin` check entirely.

---

## INFO — Liquid.sol

### Deployment Front-Run Risk on `PoolManager.initialize()`

The Liquid token relies on calling `PoolManager.initialize()` during its own initialization.

Because the token address is deterministic (factory nonce), an attacker may:

1. Predict the future token address
2. Call `PoolManager.initialize()` first
3. Cause Liquid token deployment to fail

#### Note:

SuperRare decided to address this issue in v2.

---

## INFO — Liquid.sol

### Fee Reporting Mismatch in `_disperseFees`

If a fee transfer to a creator or referrer fails:

- ETH is redirected to the protocol recipient
- But the `LiquidFees` event logs the intended, not the actual, distribution
- This creates event-log inconsistencies
- Although `FeeTransferFailed` events exist, the main event remains misleading

#### Note:

SuperRare decided to make events emit with actual fees delivered. Not intended delivery.

---

## INFO — Liquid.sol

### Uncapped Gas Forwarding Allows Creator Griefing

`_disperseFees` sends ETH using low-level calls without a gas cap, forwarding  $\frac{63}{64}$  of remaining gas.

A malicious token creator contract could:

- Consume all forwarded gas
- Leave insufficient gas to complete remaining logic
- Causing user transactions to revert
- Effectively griefing buyers/sellers of that token

#### Note:

SuperRare decided to add a “gas cap” which prevents this issue.

---

ayeslick