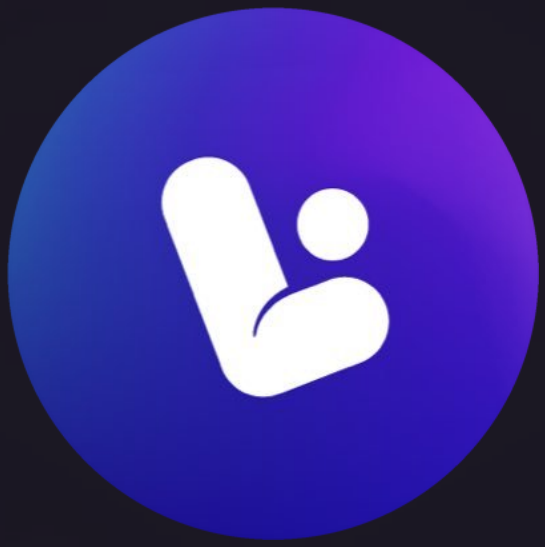
SHERLOCK

Security Review For Livo Labs



Collaborative Audit Prepared For: **Livo Labs**
Lead Security Expert(s): **0x3b**
eeyore
Date Audited: **October 14 - October 18, 2025**

Introduction

Livo Launchpad is a decentralized token launch platform that enables fair token distribution through a bonding curve mechanism. Once the launchded tokens meet a certain criteria they are graduated with automatic liquidity provision in to a decentralized exchange like Uniswap. The architecture is modular allowing different graduation mechanisms, starting with either Uniswap V2 or Uniswap V4 pools. Launching ERC20 tokens should be as simple as clicking a button, and accessible to everyone, and that's Livo's mission.

Scope

Repository: LivoLaunchpad/livo-contracts

Audited Commit: 5b476b88307ed5d53aa64ac0cfae2e4b6299d450

Final Commit: 5a1becebe3d816b37a84137c8de7da6fb4dc3b78

Files:

- src/bondingCurves/ConstantProductBondingCurve.sol
- src/graduators/LivoGraduatorUniswapV2.sol
- src/graduators/LivoGraduatorUniswapV4.sol
- src/LivoLaunchpad.sol
- src/LivoToken.sol
- src/locks/LiquidityLockUniv4WithFees.sol
- src/types/tokenData.sol

Final Commit Hash

5a1becebe3d816b37a84137c8de7da6fb4dc3b78

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or

related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	2	7

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Predictable token addresses enable DoS of token creation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/9

Summary

The `createToken()` function uses predictable clone addresses that allow attackers to precalculate future token addresses and create Uniswap pairs or pools for them, causing the legitimate token creation to revert and blocking protocol operations.

Vulnerability Detail

The `LivoLaunchpad.createToken()` function creates token clones using `Clones.clone()` which uses the `CREATE` opcode:

```
token = Clones.clone(address(tokenImplementation));
```

The `CREATE` opcode generates deterministic addresses based on the deployer address (LivoLaunchpad) and its nonce. An attacker can predict the next token address by:

1. Observing the current nonce of the `LivoLaunchpad` contract
2. Calculating the next clone address using standard `CREATE` address derivation

After predicting the address, the attacker can create the Uniswap pair/pool directly:

For UniswapV2 Graduator:

```
function initializePair(address tokenAddress) external override onlyLaunchpad
↪  returns (address pair) {
    pair = UNISWAP_FACTORY.createPair(tokenAddress, WETH); // Reverts if pair
    ↪  already exists
    emit PairInitialized(tokenAddress, pair);
}
```

The attacker calls `UNISWAP_FACTORY.createPair(predictedToken, WETH)` directly (this function is public in UniswapV2Factory), causing any future `createToken()` call to revert.

For UniswapV4 Graduator:

```
function initializePair(address tokenAddress) external override onlyLaunchpad
↪  returns (address) {
    PoolKey memory pool = _getPoolKey(tokenAddress);
    UNIV4_POOL_MANAGER.initialize(pool, SQRT_PRICEX96_GRADUATION); // Reverts if
    ↪  pool already initialized
```

```
    return address(UNIV4_POOL_MANAGER);
}
```

The attacker calls `UNIV4_POOL_MANAGER.initialize(pool, arbitraryPrice)` directly (this function is public in PoolManager and will revert on second initialization), causing any future `createToken()` call to revert.

Impact

Griefing attack that blocks token creation functionality.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L138 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/graduators/LivoGraduatorUniswapV2.sol#L52 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/graduators/LivoGraduatorUniswapV4.sol#L149

Tool Used

Manual Review

Recommendation

Implement one of the following solutions:

Option 1: Use salt-based deterministic clones

```
-function createToken(string calldata name, string calldata symbol, address
 ↪  bondingCurve, address graduator)
+function createToken(string calldata name, string calldata symbol, address
 ↪  bondingCurve, address graduator, bytes32 salt)
    external
    returns (address token)
{
    require(bytes(name).length > 0 && bytes(symbol).length > 0,
     ↪  InvalidNameOrSymbol());
    require(bytes(symbol).length <= 32, InvalidNameOrSymbol());
    require(whitelistedComponents[bondingCurve][graduator],
     ↪  InvalidCurveGraduatorCombination());

-    token = Clones.clone(address(tokenImplementation));
+    token = Clones.cloneDeterministic(address(tokenImplementation), salt);
```

```
    // ... rest of the function
}
```

Option 2: Recheck for pair initialization

```
function initializePair(address tokenAddress) external override onlyLaunchpad
 ↪  returns (address pair) {
-    pair = UNISWAP_FACTORY.createPair(tokenAddress, WETH);
+    pair = UNISWAP_FACTORY.getPair(tokenAddress, WETH);
+    if (pair == address(0)) {
+        pair = UNISWAP_FACTORY.createPair(tokenAddress, WETH);
+    }
    emit PairInitialized(tokenAddress, pair);
}
```

For UniswapV4, implement similar logic checking if the pool is already initialized before calling `initialize()`.

Issue M-2: Current tick range causes ~19% of graduation ETH to be swept to treasury [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/14

Summary

The Uniswap V4 graduation uses concentrated liquidity with tick bounds that only utilize ~80% of the ETH for liquidity provision. The remaining ~19% (approximately 1.43 ETH) is swept to the treasury, resulting in users funds being taken as additional fees beyond the explicit 0.5 ETH graduation fee.

Vulnerability Detail

During Uniswap V4 graduation, the `LivoGraduatorUniswapV4` contract creates a concentrated liquidity position with the following bounds:

```
int24 constant TICK_LOWER = -7000;
int24 constant TICK_UPPER = 203600;
int24 constant TICK_GRADUATION = 170600;
```

The liquidity is calculated using `LiquidityAmounts.getLiquidityForAmounts()`.

Due to concentrated liquidity mechanics, when both assets are provided, the function calculates liquidity from each asset independently and uses the minimum to ensure proper ratios.

In Uniswap V4, liquidity (L) is an abstract unit representing pool depth. The system calculates how much liquidity each asset could provide alone:

- **L0** (from ~7.456 ETH) ~= 4.67 * 10e22 - maximum liquidity if only ETH were deposited
- **L1** (from ~191M tokens) ~= 3.78 * 10e22 - maximum liquidity if only tokens were deposited

The minimum of the two is used to ensure both assets maintain the proper price ratio, which is **L1**. This results in:

- ETH deposited: 6.025 ETH (80.8%)
- Tokens deposited: 191.12M tokens (100%)
- **ETH leftover**: 1.43 ETH (19.2%)

The leftover ETH is then swept to the treasury via the `SWEEP` action:

```
params[2] = abi.encode(pool.currency0, excessEthRecipient); // sweep all remaining
↪  native ETH to recipient
```

Impact

Users who participated in the token sale lose approximately 19% of their contributed ETH to the treasury as unintended fees. This violates user expectations as the explicit fee structure only advertises the 0.5 ETH graduation fee.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/graduators/LivoGraduatorUniswapV4.sol#L269-L275
https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/graduators/LivoGraduatorUniswapV4.sol#L292

Tool Used

Manual Review

Recommendation

Option 1: Extend upper bound

```
- int24 constant TICK_UPPER = 203600;
+ int24 constant TICK_UPPER = 529000;
```

This achieves:

- ETH utilization: 100%
- Token utilization: 99.99%
- Maintains lower price bound

However, this removes meaningful price floor (minimum price becomes essentially zero).

Option 2: Deploy leftover ETH as single-sided liquidity

Keep the current concentrated range for the main position, but use the leftover 1.43 ETH to create a second single-sided position above the current price:

```
  // Main position (current)
  TICK_LOWER = -7000, TICK_UPPER = 203600
  Uses: 6.025 ETH + 191.1M tokens

+ // Second position (single-sided ETH only)
+ TICK_LOWER_2 = TICK_GRADUATION + TICK_SPACING  // e.g., 170800
+ TICK_UPPER_2 = TICK_UPPER                      // 203600
+ Uses: 1.43 ETH (remaining), 0 tokens
```

This achieves:

- ETH utilization: 100% (no sweep needed)
- Provides additional liquidity when token price drops (fewer tokens/ETH)
- Maintains concentrated liquidity benefits
- Maximizes capital efficiency for LPs

Issue L-1: Unprotected `initialize()` function in `LivoToken` contract enables impersonation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/8

Summary

The `LivoToken` implementation contract lacks access control on its `initialize()` function and doesn't disable initializers in the constructor, allowing anyone to initialize it and mint tokens, potentially enabling impersonation of the Livo team.

Vulnerability Detail

The `LivoToken` contract is used as an implementation for minimal proxy clones. While clones are properly initialized during creation in `LivoLaunchpad.createToken()`, the implementation contract itself can be initialized by any user:

```
function initialize(
    string memory name_,
    string memory symbol_,
    address graduator_,
    address pair_,
    address supplyReceiver_,
    uint256 totalSupply_
) external {
    require(graduator_ != address(0), InvalidGraduator());
    require(graduator == address(0), AlreadyInitialized());

    _tokenName = name_;
    _tokenSymbol = symbol_;
    graduator = graduator_;
    pair = pair_;

    _mint(supplyReceiver_, totalSupply_);
}
```

The function only checks if `graduator == address(0)`, meaning anyone can call it once on the implementation contract. The constructor also doesn't protect against this:

```
constructor() ERC20("", "") {}
```

An attacker can exploit this by:

1. Initializing the implementation contract deployed by the Livo team
2. Setting arbitrary name/symbol to impersonate legitimate Livo token

3. Minting tokens to himself
4. Creating Uniswap pairs using this token
5. Leveraging the implementation contract's association with the Livo deployer address to mislead users into thinking it's an official Livo token

Impact

Attackers can impersonate the Livo team by initializing the implementation contract, potentially causing significant reputation damage and financial loss for users who fall victim to the scam.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoToken.sol#L39 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoToken.sol#L48-L66

Tool Used

Manual Review

Recommendation

Use OpenZeppelin's `Initializable` contract to properly protect the implementation:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
+import {Initializable} from
↪  "lib/openzeppelin-contracts/contracts/proxy/utils/Initializable.sol";

-contract LivoToken is ERC20 {
+contract LivoToken is ERC20, Initializable {

-    constructor() ERC20("", "") {}
+    constructor() ERC20("", "") {
+        _disableInitializers();
+    }

    function initialize(
        string memory name_,
        string memory symbol_,
        address graduator_,
        address pair_,
```

```
        address supplyReceiver_,
        uint256 totalSupply_
-    ) external {
+    ) external initializer {
        require(graduator_ != address(0), InvalidGraduator());
-        require(graduator == address(0), AlreadyInitialized());

        _tokenName = name_;
        _tokenSymbol = symbol_;
        graduator = graduator_;
        pair = pair_;

        _mint(supplyReceiver_, totalSupply_);
    }
}
```

Issue L-2: Excess cap check incorrectly compares after-fee ETH with before-fee limit [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/10

Summary

The `quoteBuyWithExactEth()` function incorrectly compares net ETH amount (after fee deduction) with gross ETH limit (before fee deduction) when checking excess cap limits, leading to incorrect validation logic.

Vulnerability Detail

The `quoteBuyWithExactEth()` function performs an excess limit check but compares values with different fee accounting:

```
function quoteBuyWithExactEth(address token, uint256 ethValue)
    external
    view
    returns (uint256 ethForPurchase, uint256 ethFee, uint256 tokensToReceive)
{
    // ...
    if (ethForPurchase > _maxEthToSpend(token)) revert
    ↪  PurchaseExceedsLimitPostGraduation();
    // ...
}
```

The issue occurs in the comparison `ethForPurchase > _maxEthToSpend(token)`:

`ethForPurchase` calculation (from `_quoteBuyWithExactEth`):

```
ethFee = (ethValue * tokenConfig.buyFeeBps) / BASIS_POINTS;
ethForPurchase = ethValue - ethFee;
```

`_maxEthToSpend()` calculation:

```
function _maxEthToSpend(address token) internal view returns (uint256 ethBuy) {
    uint256 maxEthReserves = tokenConfigs[token].ethGraduationThreshold +
    ↪  graduationExcessCap - 1;
    uint256 remainingReserves = maxEthReserves - tokenStates[token].ethCollected;

    // apply inverse fees
    ethBuy = (remainingReserves * BASIS_POINTS) / (BASIS_POINTS -
    ↪  tokenConfigs[token].buyFeeBps);
}
```

The function compares:

- `ethForPurchase`: ETH amount **after** fee deduction (net)
- `_maxEthToSpend()`: ETH amount **before** fee deduction (gross)

This creates inconsistent validation where the limit check uses different fee accounting than what's being validated.

Impact

Incorrect validation logic.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L286 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L452-L458 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L467-L468

Tool Used

Manual Review

Recommendation

Fix the comparison to use consistent fee accounting by comparing gross amounts:

```
function quoteBuyWithExactEth(address token, uint256 ethValue)
    external
    view
    returns (uint256 ethForPurchase, uint256 ethFee, uint256 tokensToReceive)
{
+   if (ethValue > _maxEthToSpend(token)) revert
 ↪  PurchaseExceedsLimitPostGraduation();

    (ethForPurchase, ethFee, tokensToReceive) = _quoteBuyWithExactEth(token,
     ↪  ethValue);

-   if (ethForPurchase > _maxEthToSpend(token)) revert
 ↪  PurchaseExceedsLimitPostGraduation();
    if (tokensToReceive > _availableTokensForPurchase(token)) revert
     ↪  NotEnoughSupply();
}
```

Issue L-3: Reserve check incorrectly uses net seller amount instead of gross withdrawal [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/11

Summary

The `quoteSellExactTokens()` function incorrectly validates available reserves by comparing them against the net amount paid to the seller instead of the gross amount withdrawn from reserves, leading to incorrect validation logic.

Vulnerability Detail

The `quoteSellExactTokens()` function performs a reserve sufficiency check using the wrong amount:

```
function quoteSellExactTokens(address token, uint256 tokenAmount)
    external
    view
    returns (uint256 ethFromSale, uint256 ethFee, uint256 ethForSeller)
{
    // ...
    if (ethForSeller > _availableEthFromReserves(token)) revert
    ↪  InsufficientETHReserves();
}
```

The issue is in the comparison `ethForSeller > _availableEthFromReserves(token)`:

`ethForSeller` calculation (from `_quoteSellExactTokens`):

```
ethFromSale =
↪  tokenConfig.bondingCurve.sellExactTokens(tokenStates[token].ethCollected,
↪  tokenAmount);
ethFee = (ethFromSale * tokenConfig.sellFeeBps) / BASIS_POINTS;
ethForSeller = ethFromSale - ethFee;
```

Actual reserves reduction (in `sellExactTokens`):

```
tokenState.ethCollected -= ethPulledFromReserves;
```

The function compares:

- `ethForSeller`: NET ETH amount paid to seller (after fee deduction)
- `_availableEthFromReserves()`: Total available reserves

But the reserves are actually reduced by the GROSS amount (`ethFromSale`), not the net amount. This creates inconsistent validation where the quote function checks a different amount than what's actually withdrawn.

Impact

Incorrect validation logic.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L303 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L256 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L483-L486

Tool Used

Manual Review

Recommendation

Fix the comparison to use the gross amount that will be withdrawn from reserves:

```
function quoteSellExactTokens(address token, uint256 tokenAmount)
    external
    view
    returns (uint256 ethFromSale, uint256 ethFee, uint256 ethForSeller)
{
    (ethFromSale, ethFee, ethForSeller) = _quoteSellExactTokens(token, tokenAmount);

-    if (ethForSeller > _availableEthFromReserves(token)) revert
 ↪  InsufficientETHReserves();
+    if (ethFromSale > _availableEthFromReserves(token)) revert
 ↪  InsufficientETHReserves();
}
```

Issue L-4: Treasury can get DOSed from earning fees on some pools [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/12

Summary

The contract uses a **push** instead of a **pull** mechanism for fees, which could be **DoS-ed**.

Vulnerability Detail

When collecting fees, the contract first transfers funds to the **token creator** and then to the **treasury**.

```
    function collectEthFees(address[] calldata tokens) external {
        // ...
        for (uint256 i = 0; i < len; i++) {
            address token = tokens[i];
            (uint256 creatorFees, uint256 treasuryFees) = _claimFromUniswap(token);
            totalTreasuryFees += treasuryFees;

            address tokenCreator =
↪   ILivoLaunchpad(LIVO_LAUNCHPAD).getTokenCreator(token);
            _transferEth(tokenCreator, creatorFees);
        }

        address treasury = ILivoLaunchpad(LIVO_LAUNCHPAD).treasury();
        _transferEth(treasury, totalTreasuryFees);
```

If the **token creator** is a **contract** with a **fallback function** that reverts on receiving ETH, it can cause the entire transaction to revert. This would prevent the treasury from earning fees on that pool.

Note that cases where this would happen are extremely rare.

Impact

The treasury will not earn fees on that pool.

Tool Used

Manual Review

Recommendation

Consider wrapping the transfer block in a `try/catch` statement and, if it reverts, sending the funds to the treasury.

Issue L-5: Unreachable `else` case in `_addLiquidityWithPriceMatching()` would revert if executed [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/13

Summary

The `else` branch in `_addLiquidityWithPriceMatching()` is unreachable in the current code flow due to the `ethValue > 0` precondition enforced by the caller. Additionally, even if it were reachable, it would revert when calling `_naiveLiquidityAddition()` due to UniswapV2 router behavior with existing reserves.

Vulnerability Detail

The function contains defensive code for handling ETH donations to the pair:

```
function _addLiquidityWithPriceMatching(
    address tokenAddress,
    uint256 ethReserve,
    uint256 tokenBalance,
    uint256 ethValue,
    address pair
) internal returns (uint256 amountToken, uint256 amountEth, uint256 liquidity) {
    uint256 tokensToTransfer = (tokenBalance * ethReserve) / (ethValue +
    ↪  ethReserve);

    if (tokensToTransfer < tokenBalance) {
        // Normal path: transfer tokens, sync, then add remaining via router
        ...
    } else {
        // Fallback: add all as liquidity
        (amountToken, amountEth, liquidity) = _naiveLiquidityAddition(tokenAddress,
        ↪  tokenBalance, ethValue);
    }
}
```

Issue 1: Unreachable Code

The else condition `tokensToTransfer >= tokenBalance` cannot be reached due to the caller's preconditions:

```
tokensToTransfer = (tokenBalance * ethReserve) / (ethValue + ethReserve)

For tokensToTransfer >= tokenBalance:
tokenBalance * ethReserve >= tokenBalance * (ethValue + ethReserve)
```

```
ethReserve >= ethValue + ethReserve
0 >= ethValue
```

This would require `ethValue == 0`. However, `_addLiquidityWithPriceMatching()` is only called from `graduateToken()`, which enforces:

```
require(ethValue > 0, NoETHToGraduate());
```

With `ethValue > 0` guaranteed, the ratio `ethReserve / (ethValue + ethReserve)` is always < 1 for any finite donation, making the else branch unreachable in the current code flow. The else case would only trigger if this function were called with `ethValue == 0`, which cannot happen.

Issue 2: Would Revert If Reachable

Even if the else case were somehow reachable, calling `_naiveLiquidityAddition()` when `ethReserve > 0` would cause a revert:

The call chain: `_naiveLiquidityAddition()` -> `UniswapV2Router02.addLiquidityETH()` -> `_addLiquidity()` -> `UniswapV2Library.quote()`

In `UniswapV2Router02._addLiquidity()`, when reserves exist (reserveA > 0 or reserveB > 0), it calls:

```
uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA, reserveB);
```

The `quote()` function (UniswapV2Library.sol#L38) requires `reserveA > 0`:

```
function quote(uint amountA, uint reserveA, uint reserveB) internal pure returns
↪  (uint amountB) {
    require(amountA > 0, 'UniswapV2Library: INSUFFICIENT_AMOUNT');
    require(reserveA > 0 && reserveB > 0, 'UniswapV2Library:
    ↪  INSUFFICIENT_LIQUIDITY'); // <== would revert here
    amountB = amountA.mul(reserveB) / reserveA;
}
```

Since we're in `_addLiquidityWithPriceMatching()` because `ethReserve > 0` (ETH was donated), but token reserve is 0, the call to `quote()` would revert on the `require(reserveA > 0 && reserveB > 0)` check.

Impact

Informational, the code contains unreachable defensive logic that would fail if executed.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/graduators/LivoGraduatorUniswapV2.sol#L134-L151

Tool Used

Manual Review

Recommendation

Remove the unreachable else branch:

```
function _addLiquidityWithPriceMatching(...) internal returns (...) {
    uint256 tokensToTransfer = (tokenBalance * ethReserve) / (ethValue +
    ↪  ethReserve);

-   if (tokensToTransfer < tokenBalance) {
+   // Note: tokensToTransfer is always < tokenBalance due to formula (ratio always
↪   < 1)
        ILivoToken(tokenAddress).safeTransfer(pair, tokensToTransfer);
        IUniswapV2Pair(pair).sync();

        uint256 remainingTokens = tokenBalance - tokensToTransfer;
        (amountToken, amountEth, liquidity) = UNISWAP_ROUTER.addLiquidityETH{value:
        ↪  ethValue}(
            tokenAddress, remainingTokens, 0, 0, DEAD_ADDRESS, block.timestamp +
            ↪  3600
        );
        amountToken += tokensToTransfer;
-   } else {
-       (amountToken, amountEth, liquidity) = _naiveLiquidityAddition(tokenAddress,
↪   tokenBalance, ethValue);
-   }
}
```

Issue L-6: Token donation to V2 graduator deflates graduation price below bonding curve [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/15

Summary

The `LivoGraduatorUniswapV2.graduateToken()` function reads token balance directly from the contract using `token.balanceOf(address(this))`, allowing anyone to donate tokens to the graduator before graduation. This inflates the token supply added to the Uniswap V2 pool, deflating the token's graduation price below the bonding curve exit price.

Note: `LivoGraduatorUniswapV4` is not affected by this issue as the pool price is set during initialization, not determined by the token/ETH ratio at liquidity provision.

Vulnerability Detail

While the `LivoToken` contract prevents token transfers to the Uniswap pair before graduation, it does NOT prevent transfers to the graduator contract itself. The `graduateToken()` function uses the entire token balance in the graduator:

```
function graduateToken(address tokenAddress) external payable override
↪   onlyLaunchpad {
    // ...
    uint256 tokenBalance = token.balanceOf(address(this)); // includes any donations
    // ...
```

An attacker can:

1. Buy tokens on the bonding curve
2. Transfer them to the graduator contract before graduation
3. When graduation occurs, the pool is created with donated tokens + normal tokens
4. This increases the token/ETH ratio, making the token cheaper

Example calculation:

Normal graduation:

- Tokens: 191,123,251
- ETH: 7.456
- Price: 25,633,483 tokens/ETH
- Token value: $0.000156 (at ETH = 4000)$

With 50M token donation:

- Tokens: 241,123,251 (donated + normal)
- ETH: 7.456 (unchanged)
- Price: 32,339,492 tokens/ETH
- Token value: $0.000124 (\text{at} \text{ETH} = 4000)$
- **Token becomes 20.7% CHEAPER**

Impact

Token creators can be griefed by having their token launch at a deflated price, reducing the perceived value and creating a poor initial market. While the attacker loses the donated tokens, this could be used to damage specific token launches or competitors.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/graduators/LivoGraduatorUniswapV2.sol#L63

Tool Used

Manual Review

Recommendation

Pass the exact token amount from the LivoLaunchpad instead of reading the balance:

```
- function graduateToken(address tokenAddress) external payable override
 ↪  onlyLaunchpad {
+ function graduateToken(address tokenAddress, uint256 tokenAmount) external
 ↪  payable override onlyLaunchpad {
      ILivoToken token = ILivoToken(tokenAddress);

      uint256 ethValue = msg.value;
-     uint256 tokenBalance = token.balanceOf(address(this));
+     uint256 tokenBalance = tokenAmount; // Use passed amount, not balance

      require(tokenBalance > 0, NoTokensToGraduate());
      require(ethValue > 0, NoETHToGraduate());

      // ... rest of function
  }
```

Issue L-7: Global graduation parameters break price calibration when changed [RESOLVED]

Source: https://github.com/sherlock-audit/2025-10-livo-oct-14th/issues/16

Summary

The `baseEthGraduationThreshold` and `baseGraduationFee` are global variables that affect all tokens, but both graduators have hardcoded price assumptions calibrated for specific values. Admin changes to these parameters break price matching between bonding curve and Uniswap pools, creating arbitrage opportunities or graduation failures.

Vulnerability Detail

The graduation parameters are stored globally in `LivoLaunchpad`:

```
uint256 public graduationExcessCap;
uint256 public baseEthGraduationThreshold; // Default: 7956000000000052224 (~7.956
↪  ETH)
uint256 public baseGraduationFee;          // Default: 0.5 ETH
```

However, both graduators rely on these specific values:

LivoGraduatorUniswapV4 hardcodes the graduation price based on the default threshold:

```
uint160 constant SQRT_PRICEX96_GRADUATION = 401129254579132618442796085280768;
// Calibrated for 7.956 ETH threshold → ~25.6M tokens/ETH
```

LivoGraduatorUniswapV2 relies on the bonding curve price matching the V2 pool price after fee deduction:

```
uint256 ethForGraduation = ethCollected - tokenConfig.graduationEthFee;
// V2 price = tokens / ethForGraduation
```

The bonding curve formula is calibrated such that at 7.956 ETH collected with 0.5 ETH fee, the V2 pool price (~25.6M tokens/ETH) closely matches the bonding curve marginal price, preventing arbitrage.

Breaking scenarios:

1. **Changing** `baseEthGraduationThreshold`: V4 will initialize pools at the wrong price (hardcoded `SQRT_PRICEX96_GRADUATION` no longer matches actual graduation point)
2. **Changing** `baseGraduationFee`: V2 pool price shifts. For example, reducing from 0.5 ETH to 0.25 ETH makes V2 tokens 3% more expensive than bonding curve, creating profitable arbitrage (0.1 ETH last buy → +2.9% profit)

3. **Using different values for different tokens**: Current whitelisting allows multiple curve+graduator pairs, but they all use the same global graduation parameters

Impact

Admin parameter changes can create price mismatches and arbitrage opportunities.

Code Snippet

https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/LivoLaunchpad.sol#L29-L38 https://github.com/sherlock-audit/2025-10-livo-oct-14th/blob/main/livo-contracts/src/graduators/LivoGraduatorUniswapV4.sol#L82

Tool Used

Manual Review

Recommendation

Make graduation parameters configurable per (bondingCurve, graduator) pair and store them in `tokenConfigs` at creation:

```
  struct TokenConfig {
      ILivoBondingCurve bondingCurve;
      ILivoGraduator graduator;
-     uint256 graduationEthFee;
      uint256 ethGraduationThreshold;
+     uint256 graduationEthFee;
+     uint256 graduationExcessCap;
      // ... other fields
  }

+ mapping(address curve => mapping(address graduator => GraduationParams)) public
 ↪  graduationParams;

+ struct GraduationParams {
+     uint256 ethGraduationThreshold;
+     uint256 graduationFee;
+     uint256 excessCap;
+ }
```

When creating tokens, copy the parameters for that specific curve+graduator pair into `tokenConfigs`.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.