Caviar contest
Findings & Analysis Report

2023-01-26

Table of contents

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 Caviar smart contract system written in Solidity. The audit contest took place between December 12—December 19 2022.

Wardens

130 Wardens contributed reports to the Caviar contest:

1. 0x1f8b
2. 0x52
3. 0xAgro
4. 0xDave
5. 0xDecorativePineapple
6. 0xGusMcCrae
7. 0xSmartContract
8. 0xab00
9. 0xhacksmithh
10. 0xmuxyz
11. 0xxm
12. 8olidity
13. 9svR6w
14. Apocalypto (cRat1st0s, reassor, and M0ndoHEHE)
15. Aymen0909
16. BAHOZ
17. BPZ (pa6221, Bitcoinfever244, and PrasadLak)
18. Bnke0x0
19. Bobface
20. Breeje
21. CRYP70
22. Chom
23. Diana

24. ElKu
25. Franfran
26. HE1M
27. HardlyCodeMan
28. llllllll
29. JC
30. Janio
31. Jeiwan
32. JrNet
33. Junnon
34. KingNFT
35. Koolex
36. Lambda
37. Madalad
38. NoamYakov
39. RaymondFam
40. ReyAdmirado
41. Rolezn
42. SamGMK
43. SleepingBugs (Deivitto and 0xLovesleep)
44. Tointer
45. Tricko
46. UNCHAIN (Tomo, mashharuki, yawn, keit, ahayashi, sho, KazumaHamamoto, cardene, cotoneum, mugi, yosuke, masaru, kevin_katsu, junya, daikai, mabuk, mameta, kyok1st, 0xShin, hamaup, kii, and yoki)
47. UdarTeam (ahmedov and tourist)
48. Zarf
49. __141345__
50. adriro
51. ahayashi

52. akl
53. aviggiano
54. bytehat
55. c3phas
56. carlitox477
57. carrotsmuggler
58. caventa
59. cccz
60. chaduke
61. cozzetti
62. cryptonue
63. cryptostellar5
64. dic0de
65. dipp
66. eyexploit
67. fs0c
68. gz627
69. gzeon
70. h0wl
71. haku
72. hansfriese
73. helios
74. hihen
75. imare
76. immeas
77. izhelyazkov
78. kiki_dev
79. koxuan
80. ktg

81. ladboy233
82. lukris02
83. lumoswiz
84. mauricio1802
85. millersplanet
86. minhquanym
87. minhtrng
88. nicobevi
89. obront
90. oyc_109
91. pavankv
92. rajatbeladiya
93. ret2basic
94. rjs
95. rvierdiiev
96. saneryee
97. seyni
98. shung
99. supernova
100. tnevler
101. unforgiven
102. wait
103. yixxas

This contest was judged by berndartmueller.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 5

received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 25 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 31 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 Caviar contest repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 318 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 (3)

[H-01] Reentrancy in buy function for ERC777 tokens allows buying funds with considerable discount

Submitted by carlitox477, *also found by* minhquanym, gzeon, 9svR6w, Lambda, koxuan, KingNFT, cozzetti, rvierdiiev, *and* cccz

https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L95
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L137
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L172
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L203

Current implementation of functions `add`, `remove`, `buy` and `sell` first transfer fractional tokens, and then base tokens.

If this base token is ERC777 (extension of ERC20), we can call this function without updating the base token balance, but updating the fractional token balance.

Impact

Allows to drain funds of a pairs which implements an ERC-777 token.

Proof of Concept

```
    function buy(uint256 outputAmount, uint256 maxInputAmount) publi
        // *** Checks *** //

        // check that correct eth input was sent - if the baseToken
        require(baseToken == address(0) ? msg.value == maxInputAmour

        // calculate required input amount using xyk invariant
    +   @audit Use current balances
        inputAmount = buyQuote(outputAmount);

        // check that the required amount of base tokens is less tha
        require(inputAmount <= maxInputAmount, "Slippage: amount in'
```

```
    // *** Effects *** //
+   @audit Modifies just fractional balance
    // transfer fractional tokens to sender
    _transferFrom(address(this), msg.sender, outputAmount);

    // *** Interactions *** //

    if (baseToken == address(0)) {
        // refund surplus eth
        uint256 refundAmount = maxInputAmount - inputAmount;
        if (refundAmount > 0) msg.sender.safeTransferETH(refundA
    } else {

        // transfer base tokens in
+       @audit If an ERC-777 token is used, we can re call buy f
        ERC20(baseToken).safeTransferFrom(msg.sender, address(th

    }
    emit Buy(inputAmount, outputAmount);
}



function buyQuote(uint256 outputAmount) public view returns (uir
    return (outputAmount * 1000 * baseTokenReserves()) / ((fract
}
```

The buy quote is used to calculate the amount of fractional token that the user will receive, and it should be less/equal to **maxInputAmount** sent by parameter in order to achieve a successful execution of function buy.

Current buy quote can be mathematically expressed as: $\frac{outputAmount \times 1000 \times baseTokenReserves}{fractionalTokenReserves - outPutAmount} \times 997$.

Then, about sales

```
function sell(uint256 inputAmount, uint256 minOutputAmount) publ
    // *** Checks *** //

    // calculate output amount using xyk invariant
    outputAmount = sellQuote(inputAmount);
```

```
        // check that the outputted amount of fractional tokens is g
        require(outputAmount >= minOutputAmount, "Slippage: amount o

        // *** Effects *** //

        // transfer fractional tokens from sender
    +   //@audit fractional balance is updated
        _transferFrom(msg.sender, address(this), inputAmount);

        // *** Interactions *** //

        if (baseToken == address(0)) {
            // transfer ether out
            msg.sender.safeTransferETH(outputAmount);
        } else {
            // transfer base tokens out
    +       @audit If an ERC-777 token is used, we can re call sell
            ERC20(baseToken).safeTransfer(msg.sender, outputAmount);
        }

        emit Sell(inputAmount, outputAmount);
    }



        uint256 inputAmountWithFee = inputAmount * 997;
        return (inputAmountWithFee * baseTokenReserves()) / ((fracti
    }
```

Current sellQuote function can be expressed mathematically as:

$inputAmount = \frac{inputAmount \times 997 \times baseTokenReserves} {fractionalTokenReserves \times 1000 + inputAmountWithFee}$

Then we can think next scenario to drain a pair which use an ERC-777 token as base token:

1. Let's suppose the pair has 1000 base tokens(BT777) and 1000 Fractional reserve tokens (FRT)
2. The attacker call buy function, all with next inputs:

    - outputAmount = 50

- maxInputAmount = 80

3. The attacker implements a hook, that will be executed 6 times (using a counter inside a malicus contract) when a transfer is done, and call the buy function. After this 6 times the malicious contract is call again, but this times calls the sell function, doing a huge sell for the fractional reserve token obtained.

A simulation of this attack can be visualized in next table

Operation	outputAmount (FRT)	maxInputAmount (BT777)	BT777 reserve	FRT reserve	inputAmount (BT777 to pay)	inputAmount < maxInputAmount
Attaker buy 1	50	80	1000	1000	52	TRUE
Callback buy 2	50	80	1000	950	55	TRUE
Callback buy 3	50	80	1000	900	59	TRUE
Callback buy 4	50	80	1000	850	62	TRUE
Callback buy 5	50	80	1000	800	66	TRUE
Callback buy 6	50	80	1000	750	71	TRUE
Callback buy 7	50	80	1000	700	77	TRUE

The result of this operation is that the attaker/malicious contract has 350 FRT, while BT777 reserve still has 1000 and FRT reserve has 650 tokens. The success execution needs that the attacker pays 442 BT777 eventually.

To do this, the last operation of the malicious contract is calling sell function

Operation	inputAmount(BT777)	minOutputAmount	BT777 reserve	FRT reserve	outputAmount (BT777 to receive)	outputAmount > minOutputAmount
calback Sell	350	442	1000	650	536	TRUE

The result is that the attacker now controls 536 BT777, the attacker use this balance to pay the debt of 442 BT77, with a profit of 94 BT77 tokens.

Recommended Mitigation steps

Add openzeppelin nonReentrant modifier to mentioned functions, or state clear in the documentation that this protocol should not be used with ERC777 tokens.

outdoteth (Caviar) acknowledged and commented:

> Technically valid, though we don't intend to support erc777 tokens.

[H-02] Liquidity providers may lose funds when adding liquidity

Submitted by Jeiwan, *also found by* minhtrng, minhquanym, HE1M, wait, hansfriese, BAHOZ, unforgiven, 0xxm, Junnon, bytehat, UNCHAIN, carlitox477, RaymondFam, Chom, CRYP70, 9svR6w, mauricio1802, __141345__, hihen, caventa, koxuan, obront, nicobevi, shung, cccz, Bobface, *and* chaduke

Liquidity providers may lose a portion of provided liquidity in either of the pair tokens. While the `minLpTokenAmount` protects from slippage when adding liquidity, it doesn't protect from providing liquidity at different K.

Proof of Concept

The `Pair` contract is designed to receive liquidity from liquidity providers (Pair.sol#L63). First liquidity provider in a pool may provide arbitrary token amounts and set the initial price (Pair.sol#L425-L426), but all other liquidity providers must provide liquidity proportionally to current pool reserves (Pair.sol#L420-L423). Since a pool is made of two tokens and liquidity is provided in both tokens, there's a possibility for a discrepancy: token amounts may be provided in different proportions. When this happens, the smaller of the proportions is chosen to calculate the amount of LP tokens minted (Pair.sol#L420-L423):

```
// calculate amount of lp tokens as a fraction of existing reser
uint256 baseTokenShare = (baseTokenAmount * lpTokenSupply) / bas
uint256 fractionalTokenShare = (fractionalTokenAmount * lpTokenS
```

```
return Math.min(baseTokenShare, fractionalTokenShare);
```

As a result, the difference in proportions will create an excess of tokens that won't be redeemable for the amount of LP tokens minted. The excess of tokens gets, basically, donated to the pool: it'll be shared among all liquidity providers of the pool. While the `minLpTokenAmount` argument of the `add` function (Pair.sol#L63) allows liquidity providers to set the minimal amount of LP tokens they want to receive, it doesn't allow them to minimize the disproportion of token amounts or avoid it at all.

```
// test/Pair/unit.Add.t.sol

function testLockOfFunds_AUDIT() public {
    address alice = address(0x31337);
    address bob = address(0x12345);
    vm.label(alice, "alice");
    vm.label(bob, "bob");

    deal(address(usd), alice, 100e18, true);
    deal(address(usd), bob, 100e18, true);
    deal(address(p), alice, 100e18, true);
    deal(address(p), bob, 100e18, true);

    // Alice is the first liquidity provider.
    vm.startPrank(alice);
    usd.approve(address(p), type(uint256).max);
    p.add(10 ether, 10 ether, 0);
    vm.stopPrank();

    // Bob provides liquidity to the pool and sets the minimal I
    // The token amounts are deposited in different proportions,
    // one will be chosen to calculate the amount of LP tokens I
    vm.startPrank(bob);
    usd.approve(address(p), type(uint256).max);
    uint256 minLPAmount = 1e18;
    uint256 bobLPAmount = p.add(1.2 ether, 1 ether, minLPAmount)
    vm.stopPrank();

    // Bob has received the minimal LP amount he wanted.
    assertEq(bobLPAmount, minLPAmount);

    // However, after removing all his liquidity from the pool..
    (uint256 bobUSDBefore, uint256 bobFracBefore) = (usd.balance
    vm.prank(bob);
```

```
        p.remove(minLPAmount, 0, 0);
        (uint256 bobUSDAfter, uint256 bobFracAfter) = (usd.balanceOf

        // ... Bob received less USD than he deposited.
        assertEq(bobUSDAfter - bobUSDBefore, 1.018181818181818181 et
        assertEq(bobFracAfter - bobFracBefore, 1.000000000000000000
    }
```

Recommended Mitigation Steps

In the `add` function, consider calculating optimal token amounts based on the amounts specified by user, current pool reserves, and the minimal LP tokens amount specified by user. As a reference, consider this piece from the Uniswap V2 Router: UniswapV2Router02.sol#L45-L60.

outdoteth (Caviar) confirmed and commented:

> Fixed in: https://github.com/outdoteth/caviar/pull/2

> By allowing a user to specify a `minPrice` and `maxPrice` that they are willing to LP at along with the `minLpTokenAmount` that they would like to receive. The price calculation is based on this: https://github.com/outdoteth/caviar/blob/main/src/Pair.sol#L471

[H-03] First depositor can break minting of shares

Submitted by minhquanym, *also found by* Apocalypto, 0xDecorativePineapple, Franfran, dipp, rjs, ak1, Tricko, Jeiwan, unforgiven, hansfriese, BAHOZ, unforgiven, bytehat, UNCHAIN, immeas, SamGMK, fs0c, Tointer, haku, Koolex, __141345__, ElKu, rajatbeladiya, hihen, izhelyazkov, KingNFT, koxuan, 0x52, carrotsmuggler, yixxas, HE1M, supernova, cozzetti, rvierdiiev, SamGMK, aviggiano, seyni, lumoswiz, ladboy233, chaduke, cccz, *and* eyexploit

The attack vector and impact is the same as TOB-YEARN-003, where users may not receive shares in exchange for their deposits if the total asset amount has been manipulated through a large “donation”.

Proof of Concept

In `Pair.add()`, the amount of LP token minted is calculated as

```
function addQuote(uint256 baseTokenAmount, uint256 fractionalTok
    uint256 lpTokenSupply = lpToken.totalSupply();
    if (lpTokenSupply > 0) {
        // calculate amount of lp tokens as a fraction of existi
        uint256 baseTokenShare = (baseTokenAmount * lpTokenSuppl
        uint256 fractionalTokenShare = (fractionalTokenAmount *
        return Math.min(baseTokenShare, fractionalTokenShare);
    } else {
        // if there is no liquidity then init
        return Math.sqrt(baseTokenAmount * fractionalTokenAmount
    }
}
```

An attacker can exploit using these steps

1. Create and add `1 wei baseToken - 1 wei quoteToken` to the pair. At this moment, attacker is minted `1 wei LP token` because `sqrt(1 * 1) = 1`
2. Transfer large amount of `baseToken` and `quoteToken` directly to the pair, such as `1e9 baseToken - 1e9 quoteToken`. Since no new LP token is minted, `1 wei LP token` worths `1e9 baseToken - 1e9 quoteToken`.
3. Normal users add liquidity to pool will receive `0` LP token if they add less than `1e9` token because of rounding division.

```
baseTokenShare = (X * 1) / 1e9;
fractionalTokenShare = (Y * 1) / 1e9;
```

Recommended Mitigation Steps

- Uniswap V2 solved this problem by sending the first 1000 LP tokens to the zero address. The same can be done in this case i.e. when `lpTokenSupply == 0`, send the first min liquidity LP tokens to the zero address to enable share dilution.
- In `add()`, ensure the number of LP tokens to be minted is non-zero:

```
    require(lpTokenAmount != 0, "No LP minted");
```

outdoteth (Caviar) confirmed and commented:

> Fixed in: https://github.com/outdoteth/caviar/pull/3

Medium Risk Findings (5)

[M-01] Missing deadline checks allow pending transactions to be maliciously executed

Submitted by Bobface*, also found by* cozzetti

https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L63
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L107
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L147
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L182
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L275
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L294
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L310
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L323

The `Pair` contract does not allow users to submit a deadline for their action. This missing feature enables pending transactions to be maliciously executed at a later point.

Detailed description

AMMs should provide their users with an option to limit the execution of their pending actions, such as swaps or adding and removing liquidity. The most common solution is to include a deadline timestamp as a parameter (for example see Uniswap V2). If such an option is not present, users can unknowingly perform bad trades:

1. Alice wants to swap 100 fractional NFT tokens (`fTokens`) for 1 ETH and later sell the 1 ETH for 1000 DAI. She signs the transaction calling `Pair.sell` with `inputAmount = 100 fTokens` and `minOutputAmount = 0.99 ETH` to allow for some slippage.
2. The transaction is submitted to the mempool, however, Alice chose a transaction fee that is too low for miners to be interested in including her transaction in a block. The transaction stays pending in the mempool for extended periods, which could be hours, days, weeks, or even longer.
3. When the average gas fee dropped far enough for Alice's transaction to become interesting again for miners to include it, her swap will be executed. In the meantime, the price of `ETH` could have drastically changed. She will still at least get `0.99 ETH` due to `minOutputAmount` , but the `DAI` value of that output might be significantly lower. She has unknowingly performed a bad trade due to the pending transaction she forgot about.

An even worse way this issue can be maliciously exploited is through MEV:

1. The swap transaction is still pending in the mempool. Average fees are still too high for miners to be interested in it. The price of `fToken` has gone up significantly since the transaction was signed, meaning Alice would receive a lot more `ETH` when the swap is executed. But that also means that her `minOutputAmount` value is outdated and would allow for significant slippage.
2. A MEV bot detects the pending transaction. Since the outdated `minOutputAmount` now allows for high slippage, the bot sandwiches Alice, resulting in significant profit for the bot and significant loss for Alice.

The affected functions in `Pair.sol` are:

- `add()`
- `remove()`
- `buy()`

- `sell()`
- `nftAdd()`
- `nftRemove()`
- `nftBuy()`
- `nftSell()`

Recommended Mitigation Steps

Introduce a `deadline` parameter to the mentioned functions.

A word on the severity

Categorizing this issue into medium versus high was not immediately obvious. I came to the conclusion that this is a high-severity issue for the following reason:

I run an arbitrage MEV bot myself, which also tracks pending transactions in the mempool, though for another reason than the one mentioned in this report. There is a *significant* amount of pending and even dropped transactions: over `200,000` transactions that are older than one month. These transactions do all kinds of things, from withdrawing from staking contracts to sending funds to CEXs and also performing swaps on DEXs like Uniswap. This goes to show that this issue will in fact be very real, there will be very old pending transactions wanting to perform trades without a doubt. And with the prevalence of advanced MEV bots, these transactions will be exploited as described in the second example above, leading to losses for Caviar's users.

Proof of Concept

Omitted in this case, since the exploit is solely based on the fact that there is no limit on how long a transaction is allowed to be pending, which can be clearly seen when looking at the mentioned functions.

berndartmueller (judge) decreased severity to Medium

outdoteth (Caviar) commented:

> Fixed in https://github.com/outdoteth/caviar/pull/6

> Add a deadline check.

[M-02] Price will not always be 18 decimals, as expected and outlined in the comments

Submitted by obront*, also found by* cryptostellar5*,* Tricko*,* CRYP70*,* 0xmuxyz*,* koxuan*,* 8olidity*,* yixxas*,* cozzetti*,* ktg*, and* ladboy233

The `price()` function is expected to return the price of one fractional tokens, represented in base tokens, to 18 decimals of precision. This is laid out clearly in the comments:

```
/// @notice The current price of one fractional token in base tokens with 18 decimals of precision.
/// @dev Calculated by dividing the base token reserves by the fractional token reserves.
/// @return price The price of one fractional token in base tokens * 1e18.
```

However, the formula incorrectly calculates the price to be represented in whatever number of decimals the base token is in. Since there are many common base tokens (such as USDC) that will have fewer than 18 decimals, this will create a large mismatch between expected prices and the prices that result from the function.

Proof of Concept

Prices are calculated with the following formula, where `ONE = 1e18`:

```
    return (_baseTokenReserves() * ONE) / fractionalTokenReserves();
```

We know that `fractionalTokenReserves` will always be represented in 18 decimals. This means that the `ONE` and the `fractionalTokenReserves` will cancel each other out, and we are left with the `baseTokenReserves` number of decimals for the final price.

As an example:

- We have `$1000` USDC in reserves, which at 6 decimals is 1e9
- We have 1000 fractional tokens in reserve, which at 18 decimals is 1e21
- The price calculation is `1e9 * 1e18 / 1e21 = 1e6`
- While the value should be 1 token, the 1e6 will be interpreted as just 1/1e12 tokens if we expect the price to be in 1e18

Recommended Mitigation Steps

The formula should use the decimals value of the `baseToken` to ensure that the decimals of the resulting price ends up with 18 decimals as expected:

```
    return (_baseTokenReserves() * 10 ** (36 - ERC20(baseToken).deci
```

This will multiple `baseTokenReserves` by 1e18, and then additionally by the gap between 1e18 and its own decimals count, which will result in the correct decimals value for the outputted price.

outdoteth (Caviar) confirmed and commented:

> Fixed in: https://github.com/outdoteth/caviar/pull/5

> Always ensure that the exponent is 18 greater than the denominator.

[M-03] Rounding error in `buyQuote` might result in free tokens

Submitted by Zarf*, also found by* minhtrng*,* Franfran*,* Apocalypto*,* adriro*,* 0xDave*,* wait*,* unforgiven*,* Jeiwan*,* hansfriese*,* bytehat*,* UNCHAIN*,* rajatbeladiya*,* CRYP70*,* hihen*,* koxuan*,* kiki_dev*,* yixxas*, and* chaduke

In order to guarantee the contract does not become insolvent, incoming assets should be rounded up, while outgoing assets should be rounded down.

The function `buyQuote()` calculates the amount of base tokens required to buy a given amount of fractional tokens. However, this function rounds down the required

amount, which is in favor of the buyer (i.e. he/she has to provide less base tokens for the amount of receiving fractional tokens.

Depending on the amount of current token reserves and the amount of fractional tokens the user wishes to buy, it might be possible to receive free fractional tokens.

Assume the following reserve state:

- base token reserve: 0,1 WBTC (= `1e7`)
- fractional token reserve: 10.000.000 (= `1e25`)

The user wishes to buy 0,9 fractional tokens (= `9e17`). Then, the function `buyQuote()` will calculate the amount of base tokens as follows:

```
(9e17 * 1000 * 1e7) / ((1e25 - 9e17) * 997) = 0,903
```

As division in Solidity will round down, the amount results in `0` amount of base tokens required (WBTC) to buy 0,9 fractional tokens.

Impact

Using the example above, 0,9 fractional tokens is a really small amount (`0,1 BTC / 1e7 = +- $0,00017`). Moreover, if the user keeps repeating this attack, the fractional token reserve becomes smaller, which will result in a buyQuote amount of >1, after which the tokens will not be free anymore.

Additionally, as the contract incorporates a fee of 30bps, it will likely not be insolvent. The downside would be the LP holder, which will receive a fee of less than 30bps. Hence, the impact is rated as medium.

Recommended Mitigation Steps

For incoming assets, it's recommended to round up the required amount. We could use solmate's `FixedPointMathLib` library to calculate the quote and round up. This way the required amount will always at least be 1 wei:

```
function buyQuote(uint256 outputAmount) public view returns (uir
  return mulDivUp(outputAmount * 1000, baseTokenReserves(), (fra
```

```
    }
```

outdoteth (Caviar) confirmed and commented:

> Fixed in: https://github.com/outdoteth/caviar/pull/4

> Uses muldivup from solmate to round up the calculation in buyQuote.

[M-04] It's possible to swap NFT token ids without fee and also attacker can wrap unwrap all the NFT token balance of the Pair contract and steal their air drops for those token ids

Submitted by unforgiven, *also found by* imare *and* ElKu

https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L217-L243
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L248-L262

Users can `wrap()` their NFT tokens (which id is whitelisted) and receive `1e18` fractional token or they can pay `1e18` fractional token and unwrap NFT token. there is two issue here:

1. anyone can swap their NFT token id with another NFT token id without paying any fee(both ids should be whitelisted). it's swap without fee.
2. attacker can swap his NFT token(with whitelisted id) for all the NFT balance of contract and steal those NFT tokens airdrop all in one transaction.

Proof of Concept

This is `wrap()` and `unwrap()` code:

```
    function wrap(uint256[] calldata tokenIds, bytes32[][] calld
        public
        returns (uint256 fractionalTokenAmount)
```

```
{
    // *** Checks *** //

    // check that wrapping is not closed
    require(closeTimestamp == 0, "Wrap: closed");

    // check the tokens exist in the merkle root
    _validateTokenIds(tokenIds, proofs);

    // *** Effects *** //

    // mint fractional tokens to sender
    fractionalTokenAmount = tokenIds.length * ONE;
    _mint(msg.sender, fractionalTokenAmount);

    // *** Interactions *** //

    // transfer nfts from sender
    for (uint256 i = 0; i < tokenIds.length; i++) {
        ERC721(nft).safeTransferFrom(msg.sender, address(thi
    }

    emit Wrap(tokenIds);
}

function unwrap(uint256[] calldata tokenIds) public returns
    // *** Effects *** //

    // burn fractional tokens from sender
    fractionalTokenAmount = tokenIds.length * ONE;
    _burn(msg.sender, fractionalTokenAmount);

    // *** Interactions *** //

    // transfer nfts to sender
    for (uint256 i = 0; i < tokenIds.length; i++) {
        ERC721(nft).safeTransferFrom(address(this), msg.senc
    }

    emit Unwrap(tokenIds);
}
```

As you can see it’s possible to wrap one NFT token (which id is whitelisted and is in merkle tree) and unwrap another NFT token without paying fee. so Pair contract

create NFT swap without fee for users but there is no fee generated for those who wrapped and put their fractional tokens as liquidity providers. The other issue with this is that some NFT tokens air drop new NFT tokens for NFT holders by making NFT holders to call `getAirdrop()` function. attacker can use this swap functionality to get air drop token for all the NFT balance of the Pair contract. to steps to perform this attack:

1. if Pair contract is for NFT1 and baseToken1 and also merkle tree root hash is 0x0.
2. users deposited 100 NFT1 tokens to the Pair contract.
3. NFT1 decide to airdrop some new tokens for token holders and token holders need to call `nft.getAirDrop(id)` while they own the NFT id.
4. attacker would create a contract and buy one of the NFT1 tokens (attackerID1) and wrap it to receive `1e18` fractional tokens and perform this steps in the contract:
   4.1 loop through all the NFT tokens in the Pair contract balance and:
   4.2 unwrap NFT token id=i from Pair contract by paying `1e18` fractional token.
   4.3 call `nft.getAirDrop(i)` and receive the new airdrop token. (the name of the function can be other thing not exactly `getAirDrop()`)
   4.4 wrap NFT token id=i and receive `1e18` fractional token.
5. in the end attacker would unwrap attackerID1 token from Pair contract.
   so attacker was able to receive all the air drops of the NFT tokens that were in the contract address, there could be 100 or 1000 NFT tokens in the contract address and attacker can steal their air drops in one transaction(by writing a contract). those air drops belongs to all the fractional owners and contract shouldn't allow one user to take all the air drops for himself. as airdrops are common in NFT collections so this bug is critical and would happen.

also some of the NFT tokens allows users to stake some tokens for their NFT tokens and receive rewards(for example BAYC/MAYC). if a user stakes tokens for his NFT tokens then wrap those NFT tokens then it would be possible for attacker to unwrap those tokens and steal user staked amounts. in this scenario user made a risky move and wrapped NFT tokens while they have stake but as a lot of users wants to stake for their NFTs this would make them unable to use caviar protocol.

also any other action that attacker can perform by becoming the owner of the NFT token is possible by this attack and if that action can harm the NFT token holders

then attacker can harm by doing this attack and performing that action.

Tools Used

VIM

Recommended Mitigation Steps

The real solution to prevent this attack (stealing air drops) can be hard. some of the things can be done is:

- create functionality so admin can call `getAirDrop()` functions during the airdrops before attacker.
- call `getAirDrop()` (which admin specified) function before unwrapping tokens.
- make some fee for NFT token unwrapping.
- create some lock time(some days) for each wrapped NFT that in that lock time only the one who supplied that token can unwrap it.
- create some delay for unwrapping tokens and if user wants to unwrap token he would receive it after this delay.

outdoteth (Caviar) acknowledged, but disagreed with severity

berndartmueller (judge) decreased severity to Medium

[M-05] Pair price may be manipulated by direct transfers

Submitted by Jeiwan, *also found by* BPZ, ak1, Janio, hansfriese, UNCHAIN, dic0de, *and* ladboy233

https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L391
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L479-L480
https://github.com/code-423n4/2022-12-caviar/blob/0212f9dc3b6a418803dbfacda0e340e059b8aae2/src/Pair.sol#L384

An attacker may manipulate the price of a pair by transferring tokens directly to the pair. Since the `Pair` contract exposes the `price` function, it maybe be used as a price oracle in third-party integrations. Manipulating the price of a pair may allow an attacker to steal funds from such integrations.

Proof of Concept

The `Pair` contract is a pool of two tokens, a base token and a fractional token. Its main purpose is to allow users to swap the tokens at a fair price. Since the price is calculated based on the reserves of a pair, it can only be changed in two cases:

1. when initial liquidity is added: the first liquidity provider sets the price of a pool (Pair.sol#L85-L97); other liquidity providers cannot change the price (Pair.sol#L421-L423);
2. during trades: trading adds and removes tokens from a pool, ensuring the K constant invariant is respected (Pair.sol#L194-L204, Pair.sol#L161-L173).

However, the Pair contract calculates the price using the current token balances of the contract (Pair.sol#L379-L385, Pair.sol#L477-L481):

```
function baseTokenReserves() public view returns (uint256) {
    return _baseTokenReserves();
}

function _baseTokenReserves() internal view returns (uint256) {
    return baseToken == address(0)
        ? address(this).balance - msg.value // subtract the msg.
        : ERC20(baseToken).balanceOf(address(this));
}

function fractionalTokenReserves() public view returns (uint256)
    return balanceOf[address(this)];
}
```

This allows an attacker to change the price of a pool and skip the K constant invariant check that's enforced on new liquidity (Pair.sol#L421-L423).

Recommended Mitigation Steps

Consider tracking pair's reserves internally, using state variables, similarly to how Uniswap V2 does that:

- UniswapV2Pair.sol#L22-L23:

```
uint112 private reserve0;           // uses single storage slot,
uint112 private reserve1;           // uses single storage slot,
```

- UniswapV2Pair.sol#L38-L42:

```
function getReserves() public view returns (uint112 _reserve0, u
    _reserve0 = reserve0;
    _reserve1 = reserve1;
    _blockTimestampLast = blockTimestampLast;
}
```

- UniswapV2Pair.sol#L38-L42:

```
// update reserves and, on the first call per block, price accum
function _update(uint balance0, uint balance1, uint112 _reserve(
    require(balance0 <= uint112(-1) && balance1 <= uint112(-1),
    uint32 blockTimestamp = uint32(block.timestamp % 2**32);
    uint32 timeElapsed = blockTimestamp - blockTimestampLast; //
    if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
        // * never overflows, and + overflow is desired
        price0CumulativeLast += uint(UQ112x112.encode(_reserve1)
        price1CumulativeLast += uint(UQ112x112.encode(_reserve0)
    }
    reserve0 = uint112(balance0);
    reserve1 = uint112(balance1);
    blockTimestampLast = blockTimestamp;
    emit Sync(reserve0, reserve1);
}
```

minhquanym (warden) commented:

> @berndartmueller - The recommendation suggested that it should follow Uniswap V2 and add internal state balance. However, Uniswap V2 also has function

> `sync()` allowing to sync `reserve0` and `reserve1` to current token balance of contract. It means if this is an issue, it will also be an issue after UniV2 (by direct transfers and call `sync()` immediately). Please correct me if I missed something here https://github.com/Uniswap/v2-core/blob/ee547b17853e71ed4e0101ccfd52e70d5acded58/contracts/UniswapV2Pair.sol#L198

berndartmueller (judge) commented:

> @minhquanym - The specific issue demonstrated in this submission is exposing the `Pair.price` function, which is easily manipulatable by direct transfers and thus vulnerable as a price oracle. Uniswap V2, in comparison, uses the concept of a cumulative price weighted by the amount of time this price existed (see https://docs.uniswap.org/contracts/v2/concepts/core-concepts/oracles for more details).

outdoteth (Caviar) acknowledged

Low Risk and Non-Critical Issues

For this contest, 25 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: SleepingBugs, JC, rjs, minhquanym, lllllll, 0xGusMcCrae, h0wl, unforgiven, UNCHAIN, immeas, ahayashi, RaymondFam, caventa, 8olidity, yixxas, obront, shung, cozzetti, rvierdiiev, Bnke0x0, Rolezn, aviggiano, helios, *and* ladboy233.

Summary

Low Risk Issues List

Number	Issues Details	Context
[L-01]	Missing ReEntrancy Guard to `withdraw` function	1
[L-02]	Use `safeTransferOwnership` instead of `transferOwnership` function	1

Number	Issues Details	Context
[L-03]	Loss of precision due to rounding	1
[L-04]	Solmate's SafeTransferLib doesn't check whether the ERC20 contract exists	10
[L-05]	Should an airdrop token arrive on the `pair.sol` contract, it will be stuck	1

Total 5 issues

Non-Critical Issues List

Number	Issues Details	Context
[N-01]	Insufficient coverage	1
[N-02]	NatSpec comments should be increased in contracts	All Contracts
[N-03]	`Function writing` that does not comply with the `Solidity Style Guide`	All Contracts
[N-04]	Solidity compiler optimizations can be problematic	
[N-05]	For modern and more readable code; update import usages	13
[N-06]	*Lock pragmas* to specific compiler version	5
[N-07]	Use underscores for number literals	2
[N-08]	Use of bytes.concat() instead of `abi.encodePacked()`	1
[N-09]	Pragma version^0.8.17 version too recent to be trusted	All Contracts
[N-10]	Add EIP-2981 NFT Royalty Standart Support	1
[N-11]	Showing the actual values of numbers in NatSpec comments makes checking and reading code easier	2
[N-12]	Missing Event for critical parameters init and change	3

Number	Issues Details	Context
[N-13]	Add to *blacklist function*	1

Total 13 issues

Suggestions

Number	Suggestion Details	
[S-01]	Project Upgrade and Stop Scenario should be	
[S-02]	Generate perfect code headers every time	

Total 2 suggestions

[L-01] Missing ReEntrancy Guard to `withdraw` function

https://github.com/code-423n4/2022-12-caviar/blob/main/src/Pair.sol#L359-L373

Impact

Position.sol contract has no Re-Entrancy protection in `withdraw` function

```
src/Pair.sol:

 function withdraw(uint256 tokenId) public {
        // check that the sender is the caviar owner
        require(caviar.owner() == msg.sender, "Withdraw: not owr

        // check that the close period has been set
        require(closeTimestamp != 0, "Withdraw not initiated");

        // check that the close grace period has passed
        require(block.timestamp >= closeTimestamp, "Not withdraw

        // transfer the nft to the caviar owner
        ERC721(nft).safeTransferFrom(address(this), msg.sender,

        emit Withdraw(tokenId);
```

```
        }
```

If the mint was initiated by a contract, then the contract is checked for its ability to receive ERC721 tokens. Without reentrancy guard, onERC721Received will allow an attacker controlled contract to call the mint again, which may not be desirable to some parties, like allowing minting more than allowed.
https://www.paradigm.xyz/2021/08/the-dangers-of-surprising-code

Proof of Concept

If `withdraw` is msg.sender contract, it can do re-entrancy by overriding `onERC721Received` function, it doesn't seem to be a serious problem since it conforms to check-effect-interaction pattern, but this is a clear re-entry due to access to other functions and pre-emit processing. is the entracy

```
reentrancy.sol:
 function onERC721Received(
    address,
    address,
    uint256,
    bytes memory
  ) public virtual override returns (bytes4) {
    //...do something
    }
    return this.onERC721Received.selector;
  }
```

Recommended Mitigation Steps

Use Openzeppelin or Solmate Re-Entrancy pattern.
Here is a example of a re-entrancy guard

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract ReEntrancyGuard {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
```

```
            locked = true;
            _;
            locked = false;
        }
    }
```

[L-02] Use `safeTransferOwnership` instead of `transferOwnership` function

Context:

```
2 results - 2 files

src/Caviar.sol:
   4: import "solmate/auth/Owned.sol";
  12: contract Caviar is Owned {


src/LpToken.sol:
   4: import "solmate/auth/Owned.sol";
  11: contract LpToken is Owned, ERC20 {
```

Description:

`transferOwnership` function is used to change Ownership from `Owned.sol`.

Use a 2 structure transferOwnership which is safer.

`safeTransferOwnership`, use it is more secure due to 2-stage ownership transfer.

Recommendation:

Use `Ownable2Step.sol`

[Ownable2Step.sol]

[L-03] Loss of precision due to rounding

Add scalars so roundings are negligible

```
src/Pair.sol:
  390:     function price() public view returns (uint256) {
```

```
391:         return (_baseTokenReserves() * ONE) / fractionalT
392:     }
```

[L-04] Solmate's `SafeTransferLib` doesn't check whether the ERC20 contract exists

Solmate's SafeTransferLib, which is often used to interact with non-compliant/unsafe ERC20 tokens, does not check whether the ERC20 contract exists. The following code will not revert in case the token doesn't exist (yet).

This is stated in the Solmate library: https://github.com/transmissions11/solmate/blob/main/src/utils/SafeTransferLib.sol#L9

```
10 results - 1 file

src/Pair.sol:
   94              // transfer base tokens in
   95:             ERC20(baseToken).safeTransferFrom(msg.sender,
   96          }

  133             // if base token is native ETH then send ethe
  134:            msg.sender.safeTransferETH(baseTokenOutputAmo
  135         } else {
  136             // transfer base tokens to sender
  137:            ERC20(baseToken).safeTransfer(msg.sender, bas
  138         }

  168             uint256 refundAmount = maxInputAmount - input
  169:            if (refundAmount > 0) msg.sender.safeTransfer
  170         } else {
  171             // transfer base tokens in
  172:            ERC20(baseToken).safeTransferFrom(msg.sender,
  173         }

  199             // transfer ether out
  200:            msg.sender.safeTransferETH(outputAmount);
  201         } else {
  202             // transfer base tokens out
  203:            ERC20(baseToken).safeTransfer(msg.sender, out
  204         }
```

```
238           for (uint256 i = 0; i < tokenIds.length; i++) {
239:              ERC721(nft).safeTransferFrom(msg.sender, addr
240           }

258           for (uint256 i = 0; i < tokenIds.length; i++) {
259:              ERC721(nft).safeTransferFrom(address(this), m
260           }

369           // transfer the nft to the caviar owner
370:          ERC721(nft).safeTransferFrom(address(this), msg.s
371
```

Recommended Mitigation Steps

Add a contract exist control in functions;

```
pragma solidity >=0.8.0;

function isContract(address _addr) private returns (bool isContr
    isContract = _addr.code.length > 0;
}
```

[L-05] Should an airdrop token arrive on the `pair.sol` contract, it will be stuck

With the `wrap()` function, NFTs are transferred to the contract and in case of airdrop due to these NFTs, it will be stuck in the contract as there is no function to take these airdrop tokens from the contract.

Important NFT project owners are given airdrops, especially since the project includes NFTs such as BAYC, Moonbirds, Doodles, Azuki, there is a high probability of receiving Airdrops, but there is no function to withdraw incoming airdrop tokens, so airdrop tokens will be stuck in the contract.

A common method for airdrops is to collect airdrops with `claim`, so the `Pair.sol` contract can be considered upgradagable, adding a function to make `claim`.

```
src/Pair.sol:
  216      /// @return fractionalTokenAmount The amount of fract
```

```
217:     function wrap(uint256[] calldata tokenIds, bytes32[]|
218:         public
219:         returns (uint256 fractionalTokenAmount)
220:     {
221:         // *** Checks *** //
222:
223:         // check that wrapping is not closed
224:         require(closeTimestamp == 0, "Wrap: closed");
225:
226:         // check the tokens exist in the merkle root
227:         _validateTokenIds(tokenIds, proofs);
228:
229:         // *** Effects *** //
230:
231:         // mint fractional tokens to sender
232:         fractionalTokenAmount = tokenIds.length * ONE;
233:         _mint(msg.sender, fractionalTokenAmount);
234:
235:         // *** Interactions *** //
236:
237:         // transfer nfts from sender
238:         for (uint256 i = 0; i < tokenIds.length; i++) {
239:             ERC721(nft).safeTransferFrom(msg.sender, addr
240:         }
241:
242:         emit Wrap(tokenIds);
243:     }
```

Recommended Mitigation Steps

Add this code:

```
/**
 * @notice Sends ERC20 tokens trapped in contract to external a
 * @dev Onlyowner is allowed to make this function call
 * @param account is the receiving address
 * @param externalToken is the token being sent
 * @param amount is the quantity being sent
 * @return boolean value indicating whether the operation succe
 *
*/
 function rescueERC20(address account, address externalToken, u
   IERC20(externalToken).transfer(account, amount);
   return true;
```

```
    }
}
```

[N-01] Insufficient coverage

Description:
The test coverage rate of the project is 97%. Testing all functions is best practice in terms of security criteria.

```
| File                                    | % Lines          |
|-----------------------------------------|------------------|-
| src/Caviar.sol                          | 100.00% (11/11)  |
| src/LpToken.sol                         | 100.00% (2/2)    |
| src/Pair.sol                            | 100.00% (88/88)  |
| src/lib/SafeERC20Namer.sol              | 0.00% (0/38)     |
```

Due to its capacity, test coverage is expected to be 100%.

[N-02] NatSpec comments should be increased in contracts

Context:
All Contracts

Description:
It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation.
In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.
https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommendation:
NatSpec comments should be increased in contracts

[N-03] `Function writing` that does not comply with the `Solidity Style Guide`

Context:
All Contracts

Description:
Order of Functions; ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. But there are contracts in the project that do not comply with this.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html

Functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private
- within a grouping, place the view and pure functions last

[N-04] Solidity compiler optimizations can be problematic

foundry.toml:

```
1: [profile.default]
2: src = "src"
3: out = "out"
4: libs = ["lib"]
5: solc = "0.8.17"
6: optimizer_runs = 3_000
```

Description:
Protocol has enabled optional compiler optimizations in Solidity.
There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are

disabled by default, and it is unclear how many contracts in the wild actually use them.

Therefore, it is unclear how well they are being tested and exercised.
High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG.

Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of keccak256 was reported.
A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.
It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario:
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js—causes a security vulnerability in the contracts.

Recommendation:
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug. Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

[N-05] For modern and more readable code; update import usages

Context:

```
13 results - 4 files

src/Caviar.sol:
  3
  4: import "solmate/auth/Owned.sol";
  5
  6: import "./lib/SafeERC20Namer.sol";
  7: import "./Pair.sol";
```

```
  8

src/LpToken.sol:
  3
  4: import "solmate/auth/Owned.sol";
  5: import "solmate/tokens/ERC20.sol";
  6

src/Pair.sol:
   3
   4: import "solmate/tokens/ERC20.sol";
   5: import "solmate/tokens/ERC721.sol";
   6: import "solmate/utils/MerkleProofLib.sol";
   7: import "solmate/utils/SafeTransferLib.sol";
   8: import "openzeppelin/utils/math/Math.sol";
   9
  10: import "./LpToken.sol";
  11: import "./Caviar.sol";
  12

src/lib/SafeERC20Namer.sol:
  3
  4: import "openzeppelin/utils/Strings.sol";
  5
```

Description:
Solidity code is also cleaner in another way that might not be noticeable: the struct Point. We were importing it previously with global import but not using it. The Point struct `polluted the source code` with an unnecessary object we were not using because we did not need it.
This was breaking the rule of modularity and modular programming: `only import what you need` Specific imports with curly braces allow us to apply this rule better.

Recommendation:

```
import {contract1 , contract2} from "filename.sol";
```

A good example from the ArtGobblers project;

```
import {Owned} from "solmate/auth/Owned.sol";
import {ERC721} from "solmate/tokens/ERC721.sol";
import {LibString} from "solmate/utils/LibString.sol";
```

```
import {MerkleProofLib} from "solmate/utils/MerkleProofLib.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib
import {ERC1155, ERC1155TokenReceiver} from "solmate/tokens/ERC1
import {toWadUnsafe, toDaysWadUnsafe} from "solmate/utils/Signed
```

[N-06] *Lock pragmas* to specific compiler version

Description:
Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.
https://swcregistry.io/docs/SWC-103

Recommendation:
Ethereum Smart Contract Best Practices - Lock pragmas to specific compiler version.
solidity-specific/locking-pragmas

```
5 results - 4 files

src/Caviar.sol:
  1  // SPDX-License-Identifier: MIT
  2: pragma solidity ^0.8.17;
  3

src/LpToken.sol:
  1  // SPDX-License-Identifier: MIT
  2: pragma solidity ^0.8.17;
  3

src/Pair.sol:
  1  // SPDX-License-Identifier: MIT
  2: pragma solidity ^0.8.17;
  3

src/lib/SafeERC20Namer.sol:
  1  // SPDX-License-Identifier: MIT
  2: pragma solidity ^0.8.17;
```

[N-07] Use underscores for number literals

```
2 results - 1 file

src/Pair.sol:
  399:          return (outputAmount * 1000 * baseTokenReserves()

  413:          return (inputAmountWithFee * baseTokenReserves())
```

Description:
There are occasions where certain numbers have been hardcoded, either in variable or in the code itself. Large numbers can become hard to read.

Recommendation:
Consider using underscores for number literals to improve its readability.

[N-08] Use of `bytes.concat()` instead of `abi.encodePacked()`

```
1 result - 1 file

src/Pair.sol:
  473           for (uint256 i = 0; i < tokenIds.length; i++) {
  474:              bool isValid = MerkleProofLib.verify(proofs[i
```

Rather than using `abi.encodePacked` for appending bytes, since version 0.8.4, `bytes.concat()` is enabled.

Since version 0.8.4 for appending bytes, `bytes.concat()` can be used instead of `abi.encodePacked(,)`

[N-09] Pragma version^0.8.17 version too recent to be trusted.

https://github.com/ethereum/solidity/blob/develop/Changelog.md
0.8.17 (2022-09-08)

0.8.16 (2022-08-08)
0.8.15 (2022-06-15)
0.8.10 (2021-11-09)

Unexpected bugs can be reported in recent versions;
Risks related to recent releases
Risks of complex code generation changes
Risks of new language features
Risks of known bugs

Use a non-legacy and more battle-tested version
Use 0.8.10

[N-10] Add EIP-2981 NFT Royalty Standart Support

Consider adding EIP-2981 NFT Royalty Standard to the project

https://eips.ethereum.org/EIPS/eip-2981

Royalty (Copyright — EIP 2981):

- Fixed % royalties: For example, 6% of all sales go back to artists
- Declining royalties: There may be a continuous decline in sales based on time or any other variable.
- Dynamic royalties: Varies over time or sales amount
- Upgradeable royalties: Allows a legal entity or NFT owner to change any copyright
- Incremental royalties: No royalties, for example when sold for less than `$100`
- Managed royalties: Funds are owned by a DAO, imagine the recipient is a DAO treasury
- Royalties to different people: Collectors and artists can even use royalties, not specific to a particular personality

[N-11] Showing the actual values of numbers in NatSpec comments makes checking and reading code easier

```
src/Pair.sol:
  19
-  20:     uint256 public constant ONE = 1e18
+  20:     uint256 public constant ONE = 1e18;  // 1_000_000_000
-  21:     uint256 public constant CLOSE_GRACE_PERIOD = 7 days;
+  21:     uint256 public constant CLOSE_GRACE_PERIOD = 7 days;
```

[N-12] Missing Event for critical parameters init and change

Context:

```
src/Pair.sol:
  38
  39:     constructor(
  40:         address _nft,
  41:         address _baseToken,
  42:         bytes32 _merkleRoot,
  43:         string memory pairSymbol,
  44:         string memory nftName,
  45:         string memory nftSymbol
  46:     ) ERC20(string.concat(nftName, " fractional token"), s
  47:         nft = _nft;
  48:         baseToken = _baseToken; // use address(0) for nati
  49:         merkleRoot = _merkleRoot;
  50:         lpToken = new LpToken(pairSymbol);
  51:         caviar = Caviar(msg.sender);
  52:     }

src/LpToken.sol:
  11  contract LpToken is Owned, ERC20 {
  12:     constructor(string memory pairSymbol)
  13:         Owned(msg.sender)
  14:         ERC20(string.concat(pairSymbol, " LP token"), stri
  15:     {}
```

```
src/Pair.sol:
  38
  39:     constructor(
  40:         address _nft,
  41:         address _baseToken,
  42:         bytes32 _merkleRoot,
  43:         string memory pairSymbol,
```

```
44:         string memory nftName,
45:         string memory nftSymbol
46:     ) ERC20(string.concat(nftName, " fractional token"), s
47:         nft = _nft;
48:         baseToken = _baseToken; // use address(0) for nati
49:         merkleRoot = _merkleRoot;
50:         lpToken = new LpToken(pairSymbol);
51:         caviar = Caviar(msg.sender);
52:     }
```

Description:
Events help non-contract tools to track changes, and events prevent users from being surprised by changes

Recommendation:
Add Event-Emit

[N-13] Add to *blacklist function*

NFT thefts have increased recently, so with the addition of hacked NFTs to the platform, NFTs can be converted into liquidity. To prevent this, I recommend adding the blacklist function.

Marketplaces such as Opensea have a blacklist feature that will not list NFTs that have been reported theft, NFT projects such as Manifold have blacklist functions in their smart contracts.

Here is the project example; Manifold

Manifold Contract
https://etherscan.io/address/0xe4e4003afe3765aca8149a82fc064c0b125b9e5a#code

```
    modifier nonBlacklistRequired(address extension) {
        require(!_blacklistedExtensions.contains(extension), "E
        _;
    }
```

Recommended Mitigation Steps:
Add to Blacklist function and modifier.

[S-01] Project Upgrade and Stop Scenario should be

At the start of the project, the system may need to be stopped or upgraded, I suggest you have a script beforehand and add it to the documentation.
This can also be called an " EMERGENCY STOP (CIRCUIT BREAKER) PATTERN ".

https://github.com/maxwoe/solidity_patterns/blob/master/security/EmergencyStop.sol

[S-02] Generate perfect code headers every time

Description:
I recommend using header for Solidity code layout and readability

https://github.com/transmissions11/headers

```
/*////////////////////////////////////////////////////////////////
                            TESTING 123
////////////////////////////////////////////////////////////////*/
```

outdoteth (Caviar) commented:

> Great report

berndartmueller commented:

> Great report by the warden!

Gas Optimizations

For this contest, 31 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Rolezn** received the top score from the judge.

The following wardens also submitted reports: SleepingBugs, JC, Breeje, Aymen0909, c3phas, lllllll, lukris02, cryptonue, Madalad, Diana, ret2basic, tnevler, 0xAgro, carlitox477, gz627, RaymondFam, 0xSmartContract, NoamYakov, 0xab00, JrNet, HardlyCodeMan, pavankv, Bnke0x0, ReyAdmirado, oyc_109, millersplanet, saneryee, 0x1f8b, UdarTeam, *and* 0xhacksmithh.

Summary

	Issue	Contexts	Estimated Gas Saved
G-01	`<x> += <y>` Costs More Gas Than `<x> = <x> + <y>` For State Variables	3	-
G-02	`++i` / `i++` Should Be `unchecked{++i}` / `unchecked{i++}` When It Is Not Possible For Them To Overflow, As Is The Case When Used In For- And While-loops	7	245
G-03	`require()` / `revert()` Strings Longer Than 32 Bytes Cost Extra Gas	4	-
G-04	Splitting `require()` Statements That Use `&&` Saves Gas	1	9
G-05	Public Functions To External	18	-
G-06	Optimize names to save gas	3	66
G-07	Using fixed bytes is cheaper than using `string`	2	-
G-08	Superfluous event fields	1	-
G-09	`internal` functions only called once can be inlined to save gas	3	-
G-10	Setting the `constructor` to `payable`	3	39
G-11	Functions guaranteed to revert when called by normal users can be marked `payable`	2	42
G-12	Using `unchecked` blocks to save gas	1	136

Total: 48 contexts over 12 issues

[G-01] `<x> += <y>` Costs More Gas Than `<x> = <x> + <y>` For State Variables

Proof Of Concept

```
448: balanceOf[from] -= amount;
453: balanceOf[to] += amount;
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L448

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L453

```
35: charCount += uint8(b[i]);
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/lib\SafeERC20Namer.sol#L35

[G-02] `++i` / `i++` Should Be `unchecked{++i}` / `unchecked{i++}` When It Is Not Possible For Them To Overflow, As Is The Case When Used In For- And While-loops

The unchecked keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves 30-40 gas PER LOOP

Proof Of Concept

```
238: for (uint256 i = 0; i < tokenIds.length; i++) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L238

```
258: for (uint256 i = 0; i < tokenIds.length; i++) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L258

```
468: for (uint256 i = 0; i < tokenIds.length; i++) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L468

```
13: for (uint256 j = 0; j < 32; j++) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/lib\SafeERC20Namer.sol#L13

```
22: for (uint256 j = 0; j < charCount; j++) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/lib\SafeERC20Namer.sol#L22

```
33: for (uint256 i = 32; i < 64; i++) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/lib\SafeERC20Namer.sol#L33

```
39: for (uint256 i = 0; i < charCount; i++) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/lib\SafeERC20Namer.sol#L39

[G-03] `require()` / `revert()` Strings Longer Than 32 Bytes Cost Extra Gas

Proof Of Concept

```
51: require(msg.sender == pairs[nft][baseToken][merkleRoot], "Or
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Caviar.sol#L51

```
80: require(lpTokenAmount >= minLpTokenAmount, "Slippage: lp tok
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L80

```
117: require(baseTokenOutputAmount >= minBaseTokenOutputAmount,
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L117

```
120: require(fractionalTokenOutputAmount >= minFractionalTokenOu
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L120

[G-04] Splitting `require()` statements that use `&&` saves gas

Instead of using operator `&&` on a single `require`. Using a two `require` can save more gas.

i.e. for `require(version == 1 && _bytecodeHash[1] == bytes1(0), "zf");` use:

```
require(version == 1);
require(_bytecodeHash[1] == bytes1(0));
```

Proof Of Concept

```
71: require(baseTokenAmount > 0 && fractionalTokenAmount > 0, "I
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L71

[G-05] Public Functions To External

The following functions could be set external to save gas and improve code quality. External call cost is less expensive than of public functions.

Proof Of Concept

```
function create(address nft, address baseToken, bytes32 merkleRc
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Caviar.sol#L28

```
function destroy(address nft, address baseToken, bytes32 merkleF
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Caviar.sol#L49

```
function mint(address to, uint256 amount) public onlyOwner {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/LpToken.sol#L19

```
function burn(address from, uint256 amount) public onlyOwner {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/LpToken.sol#L26

```
function buy(uint256 outputAmount, uint256 maxInputAmount) publi
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L147

```
function sell(uint256 inputAmount, uint256 minOutputAmount) publ
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L182

```
    function unwrap(uint256[] calldata tokenIds) public returns (uir
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L248

```
    function nftAdd(
            uint256 baseTokenAmount,
            uint256[] calldata tokenIds,
            uint256 minLpTokenAmount,
            bytes32[][] calldata proofs
        ) public payable returns (uint256 lpTokenAmount) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L275

```
    function nftBuy(uint256[] calldata tokenIds, uint256 maxInputAmc
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L310

```
    function close() public {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L341

```
    function withdraw(uint256 tokenId) public {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L359

```
    function baseTokenReserves() public view returns (uint256) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L379

```
    function fractionalTokenReserves() public view returns (uint256)
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L383

```
    function price() public view returns (uint256) {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L390

```
    function buyQuote(uint256 outputAmount) public view returns (uir
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L398

```
    function sellQuote(uint256 inputAmount) public view returns (uir
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L406

```
    function addQuote(uint256 baseTokenAmount, uint256 fractionalTok
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L417

```
    function removeQuote(uint256 lpTokenAmount) public view returns
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L435

[G-06] Optimize names to save gas

Contracts most called functions could simply save gas by function ordering via Method ID. Calling a function at runtime will be cheaper if the function is positioned earlier in the order (has a relatively lower Method ID) because 22 gas are added to the cost of a function for every position that came before it. The caller can save on gas if you prioritize most called functions.

See more here.

Proof Of Concept

```
File: .\Projects\caviar202212\2022-12-caviar\src\Caviar.sol
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Caviar.sol

```
File: .\Projects\caviar202212\2022-12-caviar\src\LpToken.sol
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/LpToken.sol

```
File: .\Projects\caviar202212\2022-12-caviar\src\Pair.sol
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol

Recommended Mitigation Steps

Find a lower method ID name for the most called functions for example `Call()` vs. `Call1()` is cheaper by 22 gas.
For example, the function IDs in the Gauge.sol contract will be the most used; A lower method ID may be given.

[G-07] Using fixed bytes is cheaper than using `string`

As a rule of thumb, use `bytes` for arbitrary-length raw byte data and string for arbitrary-length `string` (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of `bytes1` to `bytes32` because they are much cheaper.

Proof Of Concept

```
33: string memory baseTokenSymbol = baseToken == address(0) ? "E
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Caviar.sol#L33

```
36: string memory pairSymbol = string.concat(nftSymbol, ":", bas
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Caviar.sol#L36

[G-08] Superfluous event fields

`block.number` and `block.timestamp` are added to the event information by default, so adding them manually will waste additional gas.

Proof Of Concept

```
36: event Close(uint256 closeTimestamp);
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L36

[G-09] `internal` functions only called once can be inlined to save gas

Proof Of Concept

```
463: function _validateTokenIds
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L463

```
76: function tokenSymbol
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/lib\SafeERC20Namer.sol#L76

```
87: function tokenName
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/lib\SafeERC20Namer.sol#L87

[G-10] Setting the `constructor` to `payable`

Saves ~13 gas per instance

Proof Of Concept

```
21: constructor() Owned(msg.sender)
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Caviar.sol#L21

```
11: constructor(string memory pairSymbol)
        Owned(msg.sender)
        ERC20(string.concat(pairSymbol, " LP token"), string.cor
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/LpToken.sol#L11

```
39: constructor(
        address _nft,
        address _baseToken,
        bytes32 _merkleRoot,
        string memory pairSymbol,
        string memory nftName,
        string memory nftSymbol
    ) ERC20(string.concat(nftName, " fractional token"), string.
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L39

[G-11] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier or require such as onlyOwner/onlyX 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.

Proof Of Concept

```
19: function mint(address to, uint256 amount) public onlyOwner {
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/LpToken.sol#L19

```
26: function burn(address from, uint256 amount) public onlyOwner
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/LpToken.sol#L26

Recommended Mitigation Steps

Functions guaranteed to revert when called by normal users can be marked payable.

[G-12] Using `unchecked` blocks to save gas

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block.

Proof Of Concept

```
168: uint256 refundAmount = maxInputAmount - inputAmount;
```

https://github.com/code-423n4/2022-12-caviar/tree/main/src/Pair.sol#L168

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

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