Liquis Audit Report

Jul 29, 2023

Table of Contents

Summary

This report has been prepared for Liquis smart contract, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Overview

Project Summary

Project Name	**Liquis**
Codebase	**https://github.com/liquisfi/liquis-contracts**
Commit	**733979f63458f3255cbec3dbb072ee4eca7a56d1**
Language	**Solidity**

Audit Summary

Delivery Date	**Jul 29, 2023**
Audit Methodology	**Static Analysis, Manual Review**
Total Isssues	**6**

[WP-C1] Attacker can use a malicious `_rewardPools` address to fake queued oLIT rewards and steal funds.

Critical

Issue Description

`_pids` is replaced by `_rewardPools` in order to save gas. This optimization works fine for `FlashOptionsExerciser.sol` as it's stateless. However, the `PooledOptionsExerciser` contract utilizes storage for internal accounting of users' queued oLIT rewards amounts.

For `PooledOptionsExerciser` , because `_rewardPools` can now be arbitrarily specified by the caller, an attacker can construct and deploy malicious `_rewardPools` that allow claiming and queuing any amount of `olit` rewards (updating the internal storage of `queued[user][epoch]`), and then the attacker can call `unqueue()` to steal other users' funds.

https://github.com/liquisfi/liquis-contracts/blob/cde467e5183d1764283bd103e69b7b7b921c1054/contracts/peripheral/PooledOptionsExerciser.sol#L118-L141

```
118  function claimAndQueue(
119      address[] memory _rewardPools,
120      bool _locker,
121      bool _liqLocker
122  ) external returns (uint256 amount) {
123      for (uint256 i = 0; i < _rewardPools.length; i++) {
124          // claim all the rewards, only oLIT is sent here, the rest directly to
     sender
125          amount += IBaseRewardPool(_rewardPools[i]).getRewardFor(msg.sender, true);
126      }
127
128      if (_locker) {
129          amount += IBaseRewardPool(lockerRewards).getRewardFor(msg.sender, true);
130      }
131
132      if (_liqLocker) {
133          amount += ILiqLocker(liqLocker).getRewardFor(msg.sender);
134      }
135
136      // queue claimed oLIT rewards
```

```
137     queued[msg.sender][epoch] += amount;
138     totalQueued[epoch] += amount;
139
140     emit Queued(msg.sender, epoch, amount);
141 }
```

https://github.com/liquisfi/liquis-contracts/blob/cde467e5183d1764283bd103e69b7b7b921c1054/contracts/peripheral/PooledOptionsExerciser.sol#L143-L178

```
150 function withdrawAndQueue(
151     address[] memory _rewardPools,
152     uint256[] memory _amounts,
153     bool _locker,
154     bool _liqLocker
155 ) external returns (uint256 amount) {
156     require(_rewardPools.length == _amounts.length, "array length missmatch");
157
158     for (uint256 i = 0; i < _rewardPools.length; i++) {
159         // sender will receive the Bunni LpTokens, already unwrapped
160         IRewardPool4626(_rewardPools[i]).withdraw(_amounts[i], msg.sender,
    msg.sender);
161         // claim all the rewards, only oLIT is sent here, the rest directly to
    sender
162         amount += IBaseRewardPool(_rewardPools[i]).getRewardFor(msg.sender, true);
163     }
164
165     if (_locker) {
166         amount += IBaseRewardPool(lockerRewards).getRewardFor(msg.sender, true);
167     }
168
169     if (_liqLocker) {
170         amount += ILiqLocker(liqLocker).getRewardFor(msg.sender);
171     }
172
173     // queue claimed oLIT rewards
174     queued[msg.sender][epoch] += amount;
175     totalQueued[epoch] += amount;
176
177     emit Queued(msg.sender, epoch, amount);
178 }
```

https://github.com/liquisfi/liquis-contracts/blob/cde467e5183d1764283bd103e69b7b7b921c1054/contracts/peripheral/PooledOptionsExerciser.sol#L185-L198

```
185  function unqueue(uint256 amount) external {
186      // queued balance
187      uint256 _queued = queued[msg.sender][epoch];
188
189      // revert if queued balance insufficient
190      require(amount <= _queued, "insufficient balance");
191
192      // unqueue
193      queued[msg.sender][epoch] -= amount;
194      totalQueued[epoch] -= amount;
195      IERC20(olit).safeTransfer(msg.sender, amount);
196
197      emit Unqueued(msg.sender, epoch, amount);
198  }
```

Recommendation

Consider reverting to the previous implementation: get `PoolInfo` from mapping `_pids` .

Status

✓ Fixed

[WP-L2] `FlashOptionsExerciser#executeOperation()` calculates the `maxAmountIn` for the slippage control incorrectly.

Low

Issue Description

https://github.com/liquisfi/liquis-contracts/blob/eef979d2202d986edc2e88791302bc62c6f7481f/contracts/peripheral/FlashOptionsExerciser.sol#L387-L435

```
387  function executeOperation(
388      address asset,
389      uint256 amount,
390      uint256 premium,
391      address initiator,
392      bytes calldata params
393  ) external override returns (bool) {
394      require(msg.sender == aavePool, "untrusted lender");
395      require(initiator == address(this), "untrusted initiator");
396
397      LocalVariablesFlashLoan memory vars;
398
399      (vars.olitAmount, vars.maxSlippage) = abi.decode(params, (uint256, uint256));
400
401      // exercise the olit into lit
402      IOLit(olit).exercise(vars.olitAmount, amount, address(this), block.timestamp);
403
404      // currently flashloan fee = 5, but that could vary
405      vars.amountToRepay = amount.add(premium);
406
407      IBalancerTwapOracle.OracleAverageQuery[] memory queries = new
     IBalancerTwapOracle.OracleAverageQuery[](1);
408      queries[0] = IBalancerTwapOracle.OracleAverageQuery({
409          variable: IBalancerTwapOracle.Variable.PAIR_PRICE,
410          secs: secs,
411          ago: ago
412      });
413
414      // calculate the price weth/lit in 1e18 e.g price = 1e14
415      vars.price =
     IBalancerTwapOracle(balOracle).getTimeWeightedAverage(queries)[0];
```

```
416
417     vars.amountIn = vars.amountToRepay.mul(1e18).div(vars.price);
418     // apply our accepted slippage to amountIn
419     vars.maxAmountIn =
    vars.amountIn.mul(basisOne.add(vars.maxSlippage)).div(basisOne);
420
421     vars.wethBal = IERC20(weth).balanceOf(address(this));
422     if (vars.wethBal < vars.amountToRepay) {
423         vars.amountNeeded = vars.amountToRepay.sub(vars.wethBal);
424     } // else -> amountNeeded = 0;
425
426     // swap the necessary lit into weth, swap must start with a non-zero amount in
427     if (vars.amountNeeded > 0) {
428         _balancerSwap(vars.amountNeeded, vars.maxAmountIn, IAsset(lit),
    IAsset(weth));
429     }
430
431     // repay the flashloan, aavePool will pull the tokens from the contract
432     IERC20(asset).safeIncreaseAllowance(aavePool, vars.amountToRepay);
433
434     return true;
435 }
```

`maxAmountIn` of `weth` is derived from `amountIn`, which is the original `amount.add(premium)`.

However, the actual `amountNeeded` will subtract the `wethBal`, which means that the `maxAmountIn` used for slippage control does not match the actual `weth` amount to be swapped for.

Recommendation

```
387 function executeOperation(
388     address asset,
389     uint256 amount,
390     uint256 premium,
391     address initiator,
392     bytes calldata params
393 ) external override returns (bool) {
394     require(msg.sender == aavePool, "untrusted lender");
395     require(initiator == address(this), "untrusted initiator");
```

```
396
397     LocalVariablesFlashLoan memory vars;
398
399     (vars.olitAmount, vars.maxSlippage) = abi.decode(params, (uint256, uint256));
400
401     // exercise the olit into lit
402     IOLit(olit).exercise(vars.olitAmount, amount, address(this), block.timestamp);
403
404     // currently flashloan fee = 5, but that could vary
405     vars.amountToRepay = amount.add(premium);
406
407     IBalancerTwapOracle.OracleAverageQuery[] memory queries = new
IBalancerTwapOracle.OracleAverageQuery[](1);
408     queries[0] = IBalancerTwapOracle.OracleAverageQuery({
409         variable: IBalancerTwapOracle.Variable.PAIR_PRICE,
410         secs: secs,
411         ago: ago
412     });
413
414     // calculate the price weth/lit in 1e18 e.g price = 1e14
415     vars.price =
IBalancerTwapOracle(balOracle).getTimeWeightedAverage(queries)[0];
416
417     // vars.amountIn = vars.amountToRepay.mul(1e18).div(vars.price);
418     // apply our accepted slippage to amountIn
419     // vars.maxAmountIn =
vars.amountIn.mul(basisOne.add(vars.maxSlippage)).div(basisOne);
420
421     vars.wethBal = IERC20(weth).balanceOf(address(this));
422     if (vars.wethBal < vars.amountToRepay) {
423         vars.amountNeeded = vars.amountToRepay.sub(vars.wethBal);
424     } // else -> amountNeeded = 0;
425
426     // swap the necessary lit into weth, swap must start with a non-zero amount in
427     if (vars.amountNeeded > 0) {
428         vars.amountIn = vars.amountNeeded.mul(1e18).div(vars.price);
429         vars.maxAmountIn =
vars.amountIn.mul(basisOne.add(vars.maxSlippage)).div(basisOne);
430         _balancerSwap(vars.amountNeeded, vars.maxAmountIn, IAsset(lit),
IAsset(weth));
431     }
432
433     // repay the flashloan, aavePool will pull the tokens from the contract
```

```
434         IERC20(asset).safeIncreaseAllowance(aavePool, vars.amountToRepay);
435
436         return true;
437     }
```

Status

✓ Fixed

[WP-L3] Ill-implemented slippage control

Low

Issue Description

https://github.com/liquisfi/liquis-contracts/blob/cde467e5183d1764283bd103e69b7b7b921c1054/contracts/peripheral/LitDepositorHelper.sol#L41-L43

```
41  function getMinOut(uint256 _amount, uint256 _outputBps) external view returns
    (uint256) {
42      return _getMinOut(_amount, _outputBps);
43  }
```

https://github.com/liquisfi/liquis-contracts/blob/cde467e5183d1764283bd103e69b7b7b921c1054/contracts/peripheral/BalInvestor.sol#L70-L80

```
70  function _getMinOut(uint256 amount, uint256 minOutBps) internal view returns
    (uint256) {
71      // Gets the balancer time weighted average price denominated in WETH
72      // e.g.  if 1 WETH == 0.4 BPT, bptOraclePrice == 2.5
73      uint256 bptOraclePrice = _getBptPrice(); // e.g bptOraclePrice = 3.52e14
74      uint256 pairOraclePrice = _getPairPrice(); // e.g pairOraclePrice = 0.56e14
75      uint256 bptOraclePriceInLit = (bptOraclePrice * 1e18) / pairOraclePrice; //
    e.g bptOraclePriceInLit = 6.28e18
76      // e.g. minOut = (((100e18 * 1e18) / 2.5e18) * 9980) / 10000;
77      // e.g. minout = 39.92e18
78      uint256 minOut = (((amount * 1e18) / bptOraclePriceInLit) * minOutBps) /
    10000;
79      return minOut;
80  }
```

The output of `getMinOut()` with a fixed `minOutBps` (e.g., 100 for 1%) may never be able to be executed later as it doesn't take the liquidity of the pair into consideration. With lower liquidity, the actual slippage can easily exceed `_outputBps` .

The correct approach is to dry-run the `convertLitToBpt()` function off-chain and take the

returned `bptBalance` (say if the dry-run result is 100), then adjust it based on the `slippageBps` (when 1%, the `minOut` will be 99) and use that as the `minOut` for the on-chain transaction.

The `FlashOptionsExerciser` and `PooledOptionsExerciser` which rely on the `ILitDepositorHelper.getMinOut()` will also be prone to revert or improper slippage control.

https://github.com/liquisfi/liquis-contracts/blob/cde467e5183d1764283bd103e69b7b7b921c1054/contracts/peripheral/FlashOptionsExerciser.sol#L305-L348

```
305  function withdrawAndLock(
306      address[] memory _rewardPools,
307      uint256[] memory _amounts,
308      bool _locker,
309      bool _liqLocker,
310      bool _stake,
311      uint256 _maxSlippage
312  ) external returns (uint256 claimed) {
     @@ 313,331 @@
332
333      // convert lit to liqLit, send it to sender or stake it in liqLit staking
334      // note, convert _maxSlippage to _outputBps param used in BalInvestor
335      claimed = IERC20(lit).balanceOf(address(this));
336      if (claimed > 0) _convertLitToLiqLit(claimed, basisOne.sub(_maxSlippage),
     _stake);
337  }
338
339  function _convertLitToLiqLit(
340      uint256 amount,
341      uint256 _outputBps,
342      bool _stake
343  ) internal {
344      uint256 minOut = ILitDepositorHelper(litDepositorHelper).getMinOut(amount,
     _outputBps);
345      _stake == true
346          ? ILitDepositorHelper(litDepositorHelper).depositFor(msg.sender, amount,
     minOut, true, lockerRewards)
347          : ILitDepositorHelper(litDepositorHelper).depositFor(msg.sender, amount,
     minOut, true, address(0));
348  }
```

Recommendation

1. Remove `LitDepositorHelper.getMinOut()` and the internal function `_getMinOut()`.
2. Modify `withdrawAndLock()` to accept an off-chain calculated `minOut` instead of `_maxSlippage` as the slippage control parameter.
3. The frontend should use BalancerQueriesqueryJoin to determine the value of `minOut` based on the user's preferred `slippageBps`.

Status

✓ Fixed

[WP-L4] `claim()` should also `updateReward()`

Low

Issue Description

https://github.com/liquisfi/liquis-contracts/blob/6d92f95629d2ef08dd7a517d017563266afa482f/contracts/rewards/PrelaunchRewardsPool.sol#L241-L253

```
241  function claim() external  onlyAfterDate(START_VESTING_DATE) {
242      require(isVestingUser[msg.sender], "Not vesting User");
243
244      uint256 unclaimedAmount = getClaimableLiqVesting(msg.sender);
245      if (unclaimedAmount == 0) return;
246
247      // update rewards claimed mapping
248      claimed[msg.sender] += unclaimedAmount;
249
250      rewardToken.safeTransfer(msg.sender, unclaimedAmount);
251
252      emit Claimed(msg.sender, unclaimedAmount);
253  }
```

As new rewards can be added after `START_VESTING_DATE`, `rewardPerTokenStored` can be updated. Therefore, `rewards[account]` should also be updated to ensure that the user receives all the available rewards when they call `claim()`.

However, in the current implementation, there is a lack of the `updateReward()` modifier in `claim()`. This means that the user may not be able to claim all their rewards unless they deliberately trigger the `updateReward()` with a preceding `stake()`.

Recommendation

```
241  function claim() external updateReward(msg.sender)
     onlyAfterDate(START_VESTING_DATE) {
242      require(isVestingUser[msg.sender], "Not vesting User");
243
244      uint256 unclaimedAmount = getClaimableLiqVesting(msg.sender);
```

```
245         if (unclaimedAmount == 0) return;
246
247         // update rewards claimed mapping
248         claimed[msg.sender] += unclaimedAmount;
249
250         rewardToken.safeTransfer(msg.sender, unclaimedAmount);
251
252         emit Claimed(msg.sender, unclaimedAmount);
253     }
```

Status

✓ Fixed

[WP-M6] `FlashOptionsExerciser#withdrawAndLock()` Lack of slippage control for the execution price of oLIT.

Medium

Issue Description

https://github.com/liquisfi/liquis-contracts/blob/237aa13cd3d70dba6a5cee2f0f7db5102c660326/contracts/peripheral/FlashOptionsExerciser.sol#L385-L432

```
385  function executeOperation(
386      address asset,
387      uint256 amount,
388      uint256 premium,
389      address initiator,
390      bytes calldata params
391  ) external override returns (bool) {
392      require(msg.sender == aavePool, "untrusted lender");
393      require(initiator == address(this), "untrusted initiator");
394
395      LocalVariablesFlashLoan memory vars;
396
397      (vars.olitAmount, vars.maxSlippage) = abi.decode(params, (uint256, uint256));
398
399      // exercise the olit into lit
400      IOLit(olit).exercise(vars.olitAmount, amount, address(this), block.timestamp);
401
402      // currently flashloan fee = 5, but that could vary
403      vars.amountToRepay = amount.add(premium);
404
405      IBalancerTwapOracle.OracleAverageQuery[] memory queries = new
     IBalancerTwapOracle.OracleAverageQuery[](1);
406      queries[0] = IBalancerTwapOracle.OracleAverageQuery({
407          variable: IBalancerTwapOracle.Variable.PAIR_PRICE,
408          secs: secs,
409          ago: ago
410      });
411
412      // calculate the price weth/lit in 1e18 e.g price = 1e14
413      vars.price =
     IBalancerTwapOracle(balOracle).getTimeWeightedAverage(queries)[0];
```

```
414
415     vars.wethBal = IERC20(weth).balanceOf(address(this));
416     if (vars.wethBal < vars.amountToRepay) {
417         vars.amountNeeded = vars.amountToRepay.sub(vars.wethBal);
418     } // else -> amountNeeded = 0;
419
420     // swap the necessary lit into weth, swap must start with a non-zero amount in
421     if (vars.amountNeeded > 0) {
422         vars.amountIn = vars.amountNeeded.mul(1e18).div(vars.price);
423         // apply our accepted slippage to amountIn
424         vars.maxAmountIn =
    vars.amountIn.mul(basisOne.add(vars.maxSlippage)).div(basisOne);
425         _balancerSwap(vars.amountNeeded, vars.maxAmountIn, IAsset(lit),
    IAsset(weth));
426     }
427
428     // repay the flashloan, aavePool will pull the tokens from the contract
429     IERC20(asset).safeIncreaseAllowance(aavePool, vars.amountToRepay);
430
431     return true;
432 }
```

https://github.com/liquisfi/liquis-contracts/blob/237aa13cd3d70dba6a5cee2f0f7db5102c660326/contracts/peripheral/FlashOptionsExerciser.sol#L373-L383

```
373 function _exerciseOptions(uint256 _olitAmount, uint256 _maxSlippage) internal {
374     if (_olitAmount == 0) return;
375
376     // amount of weth needed to process the olit, rounded up
377     uint256 amount = (_olitAmount * IOracle(olitOracle).getPrice()) / 1e18 + 1;
378
379     // encode _olitAmount to avoid an extra balanceOf call in next function
380     bytes memory userData = abi.encode(_olitAmount, _maxSlippage);
381
382     IPool(aavePool).flashLoanSimple(address(this), weth, amount, userData,
    referralCode);
383 }
```

https://etherscan.deth.net/address/0x9d43ccb1aD7E0081cC8A8F1fd54D16E54A637E30

```
100  function getPrice() external view override returns (uint256 price) {
101      /// ------------------------------------------------------------------------
102      /// Storage loads
103      /// ------------------------------------------------------------------------
104
105      uint256 multiplier_ = multiplier;
106      uint256 secs_ = secs;
107      uint256 ago_ = ago;
108      uint256 minPrice_ = minPrice;
109
110      /// ------------------------------------------------------------------------
111      /// Validation
112      /// ------------------------------------------------------------------------
113
114      // ensure the Balancer oracle can return a TWAP value for the specified window
115      {
116          uint256 largestSafeQueryWindow =
     balancerTwapOracle.getLargestSafeQueryWindow();
117          if (secs_ + ago_ > largestSafeQueryWindow) revert
     BalancerOracle__TWAPOracleNotReady();
118      }
119
120      /// ------------------------------------------------------------------------
121      /// Computation
122      /// ------------------------------------------------------------------------
123
124      // query Balancer oracle to get TWAP value
125      {
126          IBalancerTwapOracle.OracleAverageQuery[] memory queries = new
     IBalancerTwapOracle.OracleAverageQuery[](1);
127          queries[0] = IBalancerTwapOracle.OracleAverageQuery({
128              variable: IBalancerTwapOracle.Variable.PAIR_PRICE,
129              secs: secs_,
130              ago: ago_
131          });
132          price = balancerTwapOracle.getTimeWeightedAverage(queries)[0];
133      }
134
135      // apply multiplier to price
136      price = price.mulDivUp(multiplier_, MULTIPLIER_DENOM);
137
```

```
138     // bound price above minPrice
139     price = price < minPrice_ ? minPrice_ : price;
140 }
```

The current design only checks for the deviation of the actual prices from the oracle prices for the swap of LIT to WETH to repay the flashloan and adding BPT liquidity in `_convertLitToLiqLit()` .

There are two issues:

1. It cannot achieve the goal of controlling the deviation from the time the user sends the transaction and the actual price of execution.
2. The change of `IOracle(olitOracle).getPrice()` is not being controlled.

As a result, the user may get a much lower return than expected.

Recommendation

As the input `olitAmount` is dynamic in the context of `withdrawAndLock()` , a direct `minOut` param does not work. Instead, consider:

1. Using a `minExchangeRate` param for slippage control in `FlashOptionsExerciser#withdrawAndLock()` ;
2. Calculate the `minOut` in `_convertLitToLiqLit()` as `minExchangeRate * olitAmount` ; `ILitDepositorHelper.getMinOut()` is no longer needed;
3. The `minExchangeRate` should be calculated off-chain based on `olitAmount` and BalancerQueriesqueryJoin.
4. `balOracle` is no longer needed. The `maxAmountIn` for the swap from LIT to WETH will be all the LIT balance.

Status

✓ Fixed

[WP-I7] Wrong exercise price calculation can cause loss to the user who called `exercise()` .

Informational

Issue Description

The formula for calculating the ratio from oLIT to LIT doesn't take into account the `minPrice` of the actual exercise price in oLIT.

As a result, when the oLIT's oracle price hits the `minPrice` , whoever called `PooledOptionsExerciser#exercise()` will get a much lower LIT in return than expected due to the wrong price calculation.

https://github.com/liquisfi/liquis-contracts/blob/69a986e711546d5f2d6909a3d3db86a1a02cb6a0/contracts/peripheral/PooledOptionsExerciser.sol#L211-L222

```
211  function _exerciseAmounts() internal view returns (uint256 amountIn, uint256
     amountOut) {
212      // oLIT amount available for exercise
213      amountIn = totalQueued[epoch];
214
215      if (amountIn == 0) return (0, 0);
216
217      // oLIT execution price denominated in LIT and expressed in bps
218      uint256 price =
     uint256(IOracle(olitOracle).multiplier()).mul(basisOne.add(fee)).div(basisOne);
219
220      // amount of LIT available for claiming is exercised LIT minus execution price
221      amountOut = amountIn.sub(amountIn.mul(price).div(basisOne));
222  }
```

https://github.com/liquisfi/liquis-contracts/blob/69a986e711546d5f2d6909a3d3db86a1a02cb6a0/contracts/peripheral/PooledOptionsExerciser.sol#L237-L251

```
237  function exercise() external {
238      // compute oLIT exercise amounts
239      (uint256 amountIn, uint256 amountOut) = _exerciseAmounts();
240
241      // Update withdrawable amount for epoch
242      // note, can only exercise once for every epoch
243      totalWithdrawable[epoch] += amountOut;
244      epoch += 1;
245
246      // Transfer oLIT to caller and LIT to exerciser contract
247      IERC20(lit).safeTransferFrom(msg.sender, address(this), amountOut);
248      IERC20(olit).safeTransfer(msg.sender, amountIn);
249
250      emit Exercised(epoch - 1, amountIn, amountOut);
251  }
```

https://etherscan.deth.net/address/0x9d43ccb1aD7E0081cC8A8F1fd54D16E54A637E30

```
100  function getPrice() external view override returns (uint256 price) {
101      /// -----------------------------------------------------------------------
102      /// Storage loads
103      /// -----------------------------------------------------------------------
104
105      uint256 multiplier_ = multiplier;
106      uint256 secs_ = secs;
107      uint256 ago_ = ago;
108      uint256 minPrice_ = minPrice;
109
110      /// -----------------------------------------------------------------------
111      /// Validation
112      /// -----------------------------------------------------------------------
113
114      // ensure the Balancer oracle can return a TWAP value for the specified window
115      {
116          uint256 largestSafeQueryWindow =
     balancerTwapOracle.getLargestSafeQueryWindow();
117          if (secs_ + ago_ > largestSafeQueryWindow) revert
     BalancerOracle__TWAPOracleNotReady();
118      }
119
120      /// -----------------------------------------------------------------------
```

```
121     /// Computation
122     /// ---------------------------------------------------------------------
123
124     // query Balancer oracle to get TWAP value
125     {
126         IBalancerTwapOracle.OracleAverageQuery[] memory queries = new
    IBalancerTwapOracle.OracleAverageQuery[](1);
127         queries[0] = IBalancerTwapOracle.OracleAverageQuery({
128             variable: IBalancerTwapOracle.Variable.PAIR_PRICE,
129             secs: secs_,
130             ago: ago_
131         });
132         price = balancerTwapOracle.getTimeWeightedAverage(queries)[0];
133     }
134
135     // apply multiplier to price
136     price = price.mulDivUp(multiplier_, MULTIPLIER_DENOM);
137
138     // bound price above minPrice
139     price = price < minPrice_ ? minPrice_ : price;
140 }
```

Recommendation

Consider calculating the ratio based on the LIT oracle price: `IPriceOracle(BALANCER_POOL_TOKEN).getTimeWeightedAverage(queries)[0]` and the actual oLIT exercise price: `IOracle(olitOracle).getPrice()` .

Status

ⓘ Acknowledged

Appendix

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by WatchPug; however, WatchPug does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Smart Contract technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.