SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Olympus
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	December 5 - December 26, 2023
Prepared on:	January 29, 2024

Introduction

Olympus is building OHM, a community-owned, decentralized and censorship-resistant reserve currency that is asset-backed, deeply liquid and used widely across Web3.

Scope

Repository: OlympusDAO/bophades

Branch: rbs-v2

Commit: e0b5cd259d7a84db3a329dab3932ec8664ae1323

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
8	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hash
tvdung94
dany.armstrong90
nobody2018
nirohgo
KupiaSec
ge6a
jasonxiale
ast3ros
Arabadzhiev
rvierdiiev
Drynooo
Bauer
bin2chen
lemonmon
coffiasd
evilakela
lil.eth
CL001
0xMR0
cu5t0mPe0
shealtielanz
AuditorPraise
jovi
ZanyBonzy
0x52

Issue H-1: OlympusPrice.v2.sol#storePrice: The moving average prices are used recursively for the calculation of the moving average price.

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/55

Found by

dany.armstrong90, nirohgo, nobody2018

Summary

The moving average prices should be calculated by only oracle feed prices. But now, they are calculated by not only oracle feed prices but also moving average price recursively.

That is, the `storePrice` function uses the current price obtained from the `_getCurrentPrice` function to update the moving average price. However, in the case of `asset.useMovingAverage = true`, the `_getCurrentPrice` function computes the current price using the moving average price.

Thus, the moving average prices are used recursively to calculate moving average price, so the current prices will be obtained incorrectly.

Vulnerability Detail

`OlympusPrice.v2.sol#storePrice` function is the following.

```
    function storePrice(address asset_) public override permissioned {
        Asset storage asset = _assetData[asset_];

        // Check if asset is approved
        if (!asset.approved) revert PRICE_AssetNotApproved(asset_);

        // Get the current price for the asset
319:    (uint256 price, uint48 currentTime) = _getCurrentPrice(asset_);

        // Store the data in the obs index
        uint256 oldestPrice = asset.obs[asset.nextObsIndex];
        asset.obs[asset.nextObsIndex] = price;

        // Update the last observation time and increment the next index
        asset.lastObservationTime = currentTime;
        asset.nextObsIndex = (asset.nextObsIndex + 1) % asset.numObservations;
        // Update the cumulative observation, if storing the moving average
        if (asset.storeMovingAverage)
331:        asset.cumulativeObs = asset.cumulativeObs + price - oldestPrice;

        // Emit event
        emit PriceStored(asset_, price, currentTime);
    }
```

L319 obtain the current price for the asset by calling the `_getCurrentPrice` function and use it to update `asset.cumulativeObs` in L331. The `_getCurrentPrice` function is the following.

```
    function _getCurrentPrice(address asset_) internal view returns (uint256,
↪   uint48) {
        Asset storage asset = _assetData[asset_];

        // Iterate through feeds to get prices to aggregate with strategy
        Component[] memory feeds = abi.decode(asset.feeds, (Component[]));
        uint256 numFeeds = feeds.length;
138:    uint256[] memory prices = asset.useMovingAverage
            ? new uint256[](numFeeds + 1)
            : new uint256[](numFeeds);
        uint8 _decimals = decimals; // cache in memory to save gas
        for (uint256 i; i < numFeeds; ) {
            (bool success_, bytes memory data_) =
↪   address(_getSubmoduleIfInstalled(feeds[i].target))
                .staticcall(
                    abi.encodeWithSelector(feeds[i].selector, asset_, _decimals,
↪   feeds[i].params)
                );

            // Store price if successful, otherwise leave as zero
            // Idea is that if you have several price calls and just
            // one fails, it'll DOS the contract with this revert.
            // We handle faulty feeds in the strategy contract.
            if (success_) prices[i] = abi.decode(data_, (uint256));

            unchecked {
                ++i;
            }
        }

        // If moving average is used in strategy, add to end of prices array
160:    if (asset.useMovingAverage) prices[numFeeds] = asset.cumulativeObs /
↪   asset.numObservations;
```

```
        // If there is only one price, ensure it is not zero and return
        // Otherwise, send to strategy to aggregate
        if (prices.length == 1) {
            if (prices[0] == 0) revert PRICE_PriceZero(asset_);
            return (prices[0], uint48(block.timestamp));
        } else {
            // Get price from strategy
            Component memory strategy = abi.decode(asset.strategy, (Component));
            (bool success, bytes memory data) =
↪   address(_getSubmoduleIfInstalled(strategy.target))
                .staticcall(abi.encodeWithSelector(strategy.selector, prices,
↪   strategy.params));

            // Ensure call was successful
            if (!success) revert PRICE_StrategyFailed(asset_, data);

            // Decode asset price
            uint256 price = abi.decode(data, (uint256));

            // Ensure value is not zero
            if (price == 0) revert PRICE_PriceZero(asset_);

            return (price, uint48(block.timestamp));
        }
    }
```

As can be seen, when `asset.useMovingAverage = true`, the `_getCurrentPrice` calculates the current price `price` using the moving average price obtained by `asset.cumulativeObs / asset.numObservations` in `L160`.

So the `price` value in `L331` is obtained from not only oracle feed prices but also moving average price. Then, `storePrice` calculates the cumulative observations `asset.cumulativeObs = asset.cumulativeObs + price - oldestPrice` using the `price` which is obtained incorrectly above.

Thus, the moving average prices are used recursively for the calculation of the moving average price.

Impact

Now the moving average prices are used recursively for the calculation of the moving average price. Then, the moving average prices become more smoothed than the intention of the administrator. That is, even when the actual price fluctuations are large, the price fluctuations of `_getCurrentPrice` function will become too small.

Moreover, even though all of the oracle price feeds fails, the moving averge prices will be calculated only by moving average prices.

Thus the current prices will become incorrect. If `_getCurrentPrice` function value is miscalculated, it will cause fatal damage to the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus-web3-master/blob/main/bophades/src/modules/PRICE/OlympusPrice.v2.sol#L312-L335
https://github.com/sherlock-audit/2023-11-olympus-web3-master/blob/main/bophades/src/modules/PRICE/OlympusPrice.v2.sol#L132-L184

Tool used

Manual Review

Recommendation

When updating the current price and cumulative observations in the `storePrice` function, it should use the oracle price feeds and not include the moving average prices. So, instead of using the `asset.useMovingAverage` state variable in the `_getCurrentPrice` function, we can add a `useMovingAverage` parameter as the following.

```
>>  function _getCurrentPrice(address asset_, bool useMovingAverage) internal
↪   view returns (uint256, uint48) {
        Asset storage asset = _assetData[asset_];

        // Iterate through feeds to get prices to aggregate with strategy
        Component[] memory feeds = abi.decode(asset.feeds, (Component[]));
        uint256 numFeeds = feeds.length;
>>      uint256[] memory prices = useMovingAverage
            ? new uint256[](numFeeds + 1)
            : new uint256[](numFeeds);
        uint8 _decimals = decimals; // cache in memory to save gas
        for (uint256 i; i < numFeeds; ) {
            (bool success_, bytes memory data_) =
↪   address(_getSubmoduleIfInstalled(feeds[i].target))
                .staticcall(
                    abi.encodeWithSelector(feeds[i].selector, asset_, _decimals,
↪   feeds[i].params)
                );

            // Store price if successful, otherwise leave as zero
            // Idea is that if you have several price calls and just
            // one fails, it'll DOS the contract with this revert.
            // We handle faulty feeds in the strategy contract.
            if (success_) prices[i] = abi.decode(data_, (uint256));

            unchecked {
                ++i;
            }
        }

        // If moving average is used in strategy, add to end of prices array
>>      if (useMovingAverage) prices[numFeeds] = asset.cumulativeObs /
↪   asset.numObservations;

        // If there is only one price, ensure it is not zero and return
        // Otherwise, send to strategy to aggregate
        if (prices.length == 1) {
            if (prices[0] == 0) revert PRICE_PriceZero(asset_);
            return (prices[0], uint48(block.timestamp));
        } else {
            // Get price from strategy
            Component memory strategy = abi.decode(asset.strategy, (Component));
            (bool success, bytes memory data) =
↪   address(_getSubmoduleIfInstalled(strategy.target))
                .staticcall(abi.encodeWithSelector(strategy.selector, prices,
↪   strategy.params));

            // Ensure call was successful
            if (!success) revert PRICE_StrategyFailed(asset_, data);

            // Decode asset price
            uint256 price = abi.decode(data, (uint256));

            // Ensure value is not zero
            if (price == 0) revert PRICE_PriceZero(asset_);

            return (price, uint48(block.timestamp));
        }
    }
```

Then we should set `useMovingAverage = false` to call `_getCurrentPrice` function only in the `storePrice` function. In other cases, we should set `useMovingAverage = asset.useMovingAverage` to call `_getCurrentPrice` function.

Discussion

0xrusowsky

https://github.com/OlympusDAO/bophades/pull/257

IAm0x52

Fix looks good. The moving average is no longer included when storing price

Issue H-2: Incorrect ProtocolOwnedLiquidityOhm calculation due to inclusion of other user's reserves

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/172

Found by

hash, tvdung94

Summary

ProtocolOwnedLiquidityOhm for Bunni can include the liquidity deposited by other users which is not protocol owned

Vulnerability Detail

The protocol owned liquidity in Bunni is calculated as the sum of reserves of all the BunniTokens

```
function getProtocolOwnedLiquidityOhm() external view override returns (uint256)
↪  {

    uint256 len = bunniTokens.length;
    uint256 total;
    for (uint256 i; i < len; ) {
        TokenData storage tokenData = bunniTokens[i];
        BunniLens lens = tokenData.lens;
        BunniKey memory key = _getBunniKey(tokenData.token);

    .........

        total += _getOhmReserves(key, lens);
        unchecked {
            ++i;
        }
    }


    return total;
}
```

The deposit function of Bunni allows any user to add liquidity to a token. Hence the returned reserve will contain amounts other than the reserves that actually belong to the protocol

```
// @audit callable by any user
function deposit(
    DepositParams calldata params
)
    external
    payable
    virtual
    override
    checkDeadline(params.deadline)
    returns (uint256 shares, uint128 addedLiquidity, uint256 amount0, uint256
↪   amount1)
{
}
```

Impact

Incorrect assumption of the protocol owned liquidity and hence the supply. An attacker can inflate the liquidity reserves The wider system relies on the supply calculation to be correct in order to perform actions of economical impact

```
https://discord.com/channels/812037309376495636/1184355501258047488/118439790455⌋
↪   1628831
it will be determined to get backing
so it will have an economical impact, as we could be exchanging ohm for treasury
↪   assets at a wrong price
```

Code Snippet

POL liquidity is calculated as the sum of bunni token reserves https://github.com/sherlock-audit/2023-11-olympus/blob/9c8df76dc9820b4c6605d2e1e6d87dcfa9e50070/bophades/src/modules/SPPLY/submodules/BunniSupply.sol#L171-L191

BunniHub allows any user to deposit https://github.com/sherlock-audit/2023-11-olympus/blob/9c8df76dc9820b4c6605d2e1e6d87dcfa9e50070/bophades/src/external/bunni/BunniHub.sol#L71-L106

Tool used

Manual Review

Recommendation

Guard the deposit function in BunniHub or compute the liquidity using shares belonging to the protocol

Discussion

0xJem

This is a good catch, and the high level is justified

0xrusowsky

https://github.com/OlympusDAO/bophades/pull/260

IAm0x52

Fix looks good. OnlyOwner modifier has been added to deposits

Issue H-3: Incorrect StablePool BPT price calculation

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/176

Found by

Bauer, ast3ros, ge6a, hash, jasonxiale, tvdung94

Summary

Incorrect StablePool BPT price calculation as rate's are not considered

Vulnerability Detail

The price of a stable pool BPT is computed as:

> minimum price among the pool tokens obtained via feeds * return value of `getRate()`

This method is used referring to an old documentation of Balancer

```
function getStablePoolTokenPrice(
    address,
    uint8 outputDecimals_,
    bytes calldata params_
) external view returns (uint256) {
    // Prevent overflow
    if (outputDecimals_ > BASE_10_MAX_EXPONENT)
        revert Balancer_OutputDecimalsOutOfBounds(outputDecimals_,
↪   BASE_10_MAX_EXPONENT);


    address[] memory tokens;
    uint256 poolRate; // pool decimals
    uint8 poolDecimals;
    bytes32 poolId;
    {

    ......

        // Get tokens in the pool from vault
        (address[] memory tokens_, , ) = balVault.getPoolTokens(poolId);
        tokens = tokens_;

        // Get rate
        try pool.getRate() returns (uint256 rate_) {
            if (rate_ == 0) {
                revert Balancer_PoolStableRateInvalid(poolId, 0);
            }


            poolRate = rate_;

    ......

    uint256 minimumPrice; // outputDecimals_
    {
        /**
         * The Balancer docs do not currently state this, but a historical version noted
         * that getRate() should be multiplied by the minimum price of the tokens in the
         * pool in order to get a valuation. This is the same approach as used by Curve stable pools.
         */
        for (uint256 i; i < len; i++) {
            address token = tokens[i];
            if (token == address(0)) revert Balancer_PoolTokenInvalid(poolId, i, token);

            (uint256 price_, ) = _PRICE().getPrice(token, PRICEv2.Variant.CURRENT); // outputDecimals_


            if (minimumPrice == 0) {
                minimumPrice = price_;
            } else if (price_ < minimumPrice) {
                minimumPrice = price_;
            }
        }
    }

    uint256 poolValue = poolRate.mulDiv(minimumPrice, 10 ** poolDecimals); // outputDecimals_
```

The `getRate()` function returns the exchange rate of a BPT to the underlying base asset of the pool which can be different from the minimum market priced asset for pools with rateProviders. To consider this, the price obtained from feeds must be divided by the `rate` provided by `rateProviders` before choosing the minimum as mentioned in the previous version of Balancer's documentation.

https://github.com/balancer/docs/blob/663e2f4f2c3eee6f85805e102434629633af92a2/docs/concepts/advanced/valuing-bpt/bpt-as-collateral.md#metastablepool s-eg-wsteth-weth

1. Get market price for each constituent token Get market price of wstETH and WETH in terms of USD, using chainlink oracles.

2. Get RateProvider price for each constituent token Since wstETH - WETH pool is a MetaStablePool and not a ComposableStablePool, it does not have `getTokenRate()` function. Therefore, it's needed to get the RateProvider price manually for wstETH, using the rate providers of the pool. The rate provider will return the wstETH token in terms of stETH.

Note that WETH does not have a rate provider for this pool. In that case, assume a value of `1e18` (it means, market price of WETH won't be divided by any value, and it's used purely in the minPrice formula).

3. Get minimum price

$$minPrice = min(\frac{P_{M_{wstETH}}}{P_{RP_{wstETH}}}, P_{M_{WETH}})$$

4. Calculates the BPT price

$$P_{BPT_{wstETH-WETH}} = minPrice * rate_{pool_{wstETH-WETH}}$$

where `rate_pool_wstETH-WETH` is `pool.getRate()` of wstETH-WETH pool.

Example

The wstEth-cbEth pool is a MetaStablePool having rate providers for both tokens since neither of them is the base token https://app.balancer.fi/#/ethereum/pool/0x9c6d47ff73e0f5e51be5fd53236e3f595c5793f200020000000000000000042c

At block 18821323: cbeth : 2317.48812 wstEth : 2526.84 pool total supply : 0.273259897168240633 getRate() : 1.022627523581711856 wstRateprovider rate : 1.150725009180224306 cbEthRateProvider rate : 1.058783029570983377 wstEth balance : 0.133842314907166538 cbeth balance : 0.119822100236557012 tvl : (0.133842314907166538 * 2526.84 + 0.119822100236557012 * 2317.48812) == 615.884408812

according to current implementation: bpt price = 2317.48812 * 1.022627523581711856 == 2369.927137086 calculated tvl = bpt price * total supply = 647.606045776

correct calculation: rate_provided_adjusted_cbeth = (2317.48812 / 1.058783029570983377) == 2188.822502132 rate_provided_adjusted_wsteth = (2526.84 / 1.150725009180224306) == 2195.867804942 bpt price =

2188.822502132 * 1.022627523581711856 == 2238.350134915 calculated tvl = bpt price * total supply = (2238.350134915 * 0.273259897168240633) == 611.651327693

Impact

Incorrect calculation of bpt price. Has possibility to be over and under valued.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/PRICE/submodules/feeds/BalancerPoolTokenPrice.sol#L514-L539

Tool used

Manual Review

Recommendation

For pools having rate providers, divide prices by rate before choosing the minimum

Discussion

0xJem

This is a valid issue and highlights problems with Balancer's documentation.

We are likely to drop both the Balancer submodules from the final version, since we no longer have any Balancer pools used for POL and don't have any assets that require price resolution via Balancer pools.

sherlock-admin2

> Escalate
>
> This is invalid. Never meant to interact with composable stable pools as shown by this comment here where they explicitly state it will revert if it is a composable stable pool:
>
> https://github.com/sherlock-audit/2023-11-olympus/blob/9c8df76dc9820b4c6605d2e1e6d87dcfa9e50070/bophades/src/modules/PRICE/submodules/feeds/BalancerPoolTokenPrice.sol#L441-L444

```
You've deleted an escalation for this issue.
```

Issue H-4: getBunniTokenPrice wrongly returns the total price of all tokens

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/198

Found by

Arabadzhiev, Drynooo, ge6a, hash, jasonxiale, tvdung94

Summary

The function getBunniTokenPrice() is supposed to return the price of 1 Bunni token (1 share) like all other feeds, but it doesn't. It returns the total price of all minted tokens/shares for a specific pool (total value of position's reserves) which is wrong.

Vulnerability Detail

This happens because the totalValue on line 163 is not devided by the total tokens supply.

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/PRICE/submodules/feeds/BunniPrice.sol#L110-L166

Impact

The function getBunniTokenPrice always returns wrong price. This would impact the operation of the RBS module. For instance, using the wrong price during a swap may lead to financial losses for the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/PRICE/submodules/feeds/BunniPrice.sol#L110-L166

Tool used

Manual Review

Recommendation

Devide totalValue by the total tokens supply.

Discussion

0xrusowsky

https://github.com/OlympusDAO/bophades/pull/244

IAm0x52

Fix looks good. Token price is now normalized to get price per token.

Issue M-1: BunniPrice.getBunniTokenPrice doesn't include fees

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/37

Found by

dany.armstrong90, rvierdiiev, tvdung94

Summary

BunniPrice.getBunniTokenPrice doesn't include fees into calculation, which means that price will be smaller.

Vulnerability Detail

`BunniPrice.getBunniTokenPrice` function should return usd cost of `bunniToken_`. After initial checks this function actually do 2 things:

- validate deviation
- calculates total cost of token

Let's check how it's done to calculate total cost of token. It's done using _getTotalValue function.

So we get token reserves using _getBunniReserves function and the convert them to usd value and return the sum. `_getBunniReserves` in it's turn just fetches amount of token0 and token1 that can be received in case if you swap all bunni token liquidity right now in the pool.

The problem is that this function doesn't include fees that are earned by the position. As result total cost of bunni token is smaller than in reality. The difference depends on the amount of fees there were earned and not returned back.

Also when you do burn in the uniswap v3, then amount of token0/token1 is not send back to the caller, but they are added to the position and can be collected later. So in case if `BunniPrice.getBunniTokenPrice` will be called for the token, where some liquidity is burnt but not collected, then difference in real token price can be huge.

Impact

Price for the bunni token can be calculated in wrong way.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Include collected amounts and additionally earned fees to the position reserves.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

nirohgo commented:

> duplicate of 137

0xJem

IMO this is medium or low severity - the fees will be compounded regularly, and so aren't going to increasing to a significant amount.

nevillehuang

Hi @0xJem, since this could occur naturally without any external factors, I think it could constitute high severity given important price values could be directly affected.

Does the following mean uncollected fees are going to be collected by olympus regularly?

> the fees will be compounded regularly

0xJem

> Hi @0xJem, since this could occur naturally without any external factors, I think it could constitute high severity given important price values could be directly affected.
>
> Does the following mean uncollected fees are going to be collected by olympus regularly?
>
> > the fees will be compounded regularly

Yes we will have a keeper function calling the harvest() function on BunniManager. It can be called maximum once in 24 hours.

0xrusowsky

https://github.com/OlympusDAO/bophades/pull/244

IAm0x52

Escalate

This is simply another occurrence of #49 in a different location and should be grouped with it

sherlock-admin2

> Escalate
>
> This is simply another occurrence of #49 in a different location and should be grouped with it

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

Agree with @IAm0x52 , root cause is the same although with different impact, can be duplicated.

Arabadzhiew

This issue and its duplicates refer to an issue in the `BunniPrice` contract, while issue #49 and its duplicates refer to an issue that is inside of the `BunniSupply` contract. The root causes are clearly not the same, so I believe those two issues should stay separated in 2 different families.

0xJem

IMO it is separate. 49 refers to the TWAP and reserves check, whereas this is valuing the LP position.

IAm0x52

Fix looks good. Fees are now also accounted for here as well.

Czar102

@IAm0x52 could you take another look and let me know if you agree with the above comments?

nevillehuang

@Czar102 I think there was a separate fix with the 246 pull request for issue #49, so could indicate that they are not duplicates

Czar102

Result: Medium Has duplicates

Escalation's author hasn't provided enough information to justify a change in status of the issue.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- IAm0x52: rejected

Issue M-2: Inconsistency in BunniToken Price Calculation

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/49

Found by

Arabadzhiev, KupiaSec, dany.armstrong90, hash, lil.eth, rvierdiiev

Summary

The deviation check (`_validateReserves()`) from BunniPrice.sol considers both position reserves and uncollected fees when validating the deviation with TWAP, while the final price calculation (`_getTotalValue()`) only accounts for position reserves, excluding uncollected fees.

The same is applied to BunniSupply.sol where `getProtocolOwnedLiquidityOhm()` validates reserves + fee deviation from TWAP and then returns only Ohm reserves using `lens_.getReserves(key_)`

Note that `BunniSupply.sol#getProtocolOwnedLiquidityReserves()` validates deviation using reserves+fees with TWAP and then return reserves+fees in a good way without discrepancy.

But this could lead to a misalignment between the deviation check and actual price computation.

Vulnerability Detail

1. Deviation Check : `_validateReserves` Function:

```
### BunniPrice.sol and BunniSupply.sol :
    function _validateReserves( BunniKey memory key_,BunniLens lens_,uint16
 ↪  twapMaxDeviationBps_,uint32 twapObservationWindow_) internal view
        {
        uint256 reservesTokenRatio = BunniHelper.getReservesRatio(key_, lens_);
        uint256 twapTokenRatio = UniswapV3OracleHelper.getTWAPRatio(address(key_
 ↪  .pool),twapObservationWindow_);

        // Revert if the relative deviation is greater than the maximum.
        if (
            // `isDeviatingWithBpsCheck()` will revert if `deviationBps` is
 ↪  invalid.
            Deviation.isDeviatingWithBpsCheck(
                reservesTokenRatio,
                twapTokenRatio,
        twapMaxDeviationBps_,
        TWAP_MAX_DEVIATION_BASE
      )
    ) {
      revert BunniPrice_PriceMismatch(address(key_.pool), twapTokenRatio,
↪  reservesTokenRatio);
    }
  }

### BunniHelper.sol :
  function getReservesRatio(BunniKey memory key_, BunniLens lens_) public view
↪  returns (uint256) {
    IUniswapV3Pool pool = key_.pool;
    uint8 token0Decimals = ERC20(pool.token0()).decimals();

    (uint112 reserve0, uint112 reserve1) = lens_.getReserves(key_);

    //E compute fees and return values
    (uint256 fee0, uint256 fee1) = lens_.getUncollectedFees(key_);

    //E calculates ratio of token1 in token0
    return (reserve1 + fee1).mulDiv(10 ** token0Decimals, reserve0 + fee0);
  }

### UniswapV3OracleHelper.sol :
  //E Returns the ratio of token1 to token0 in token1 decimals based on the
↪  TWAP
    //E used in bophades/src/modules/PRICE/submodules/feeds/BunniPrice.sol,
↪  and SPPLY/submodules/BunniSupply.sol
  function getTWAPRatio(
    address pool_,
    uint32 period_ //E period of the TWAP in seconds
  ) public view returns (uint256)
  {
    //E return the time-weighted tick from period_ to now
    int56 timeWeightedTick = getTimeWeightedTick(pool_, period_);

    IUniswapV3Pool pool = IUniswapV3Pool(pool_);
    ERC20 token0 = ERC20(pool.token0());
    ERC20 token1 = ERC20(pool.token1());

    // Quantity of token1 for 1 unit of token0 at the time-weighted tick
    // Scale: token1 decimals
    uint256 baseInQuote = OracleLibrary.getQuoteAtTick(
      int24(timeWeightedTick),
      uint128(10 ** token0.decimals()), // 1 unit of token0 => baseAmount
      address(token0),
      address(token1)
    );
    return baseInQuote;
}
```

You can see that the deviation check includes uncollected fees in the `reservesTokenRatio`, potentially leading to a higher or more volatile ratio compared to the historical `twapTokenRatio`.

2. Final Price Calculation in `BunniPrice.sol#_getTotalValue()` :

```
function _getTotalValue(
    BunniToken token_,
    BunniLens lens_,
    uint8 outputDecimals_
) internal view returns (uint256) {
    (address token0, uint256 reserve0, address token1, uint256 reserve1) =
↪   _getBunniReserves(
        token_,
        lens_,
        outputDecimals_
    );
    uint256 outputScale = 10 ** outputDecimals_;

    // Determine the value of each reserve token in USD
    uint256 totalValue;
    totalValue += _PRICE().getPrice(token0).mulDiv(reserve0, outputScale);
    totalValue += _PRICE().getPrice(token1).mulDiv(reserve1, outputScale);

    return totalValue;
}
```

You can see that this function (`_getTotalValue()`) excludes uncollected fees in the final valuation, potentially overestimating the total value within deviation check process, meaning the check could pass in certain conditions whereas it could have not pass if fees where not accounted on the deviation check. Moreover the below formula used :

$$price_{LP} = reserve_0 \times price_0 + reserve_1 \times price_1$$

where $reserve_i$ is token i reserve amount, $price_i$ is the price of token i

In short, it is calculated by getting all underlying balances, multiplying those by their market prices

However, this approach of directly computing the price of LP tokens via spot reserves is well-known to be vulnerable to manipulation, even if TWAP Deviation is checked, the above summary proved that this method is not 100% bullet proof as there are discrepancy on what is mesured. Taken into the fact that the process to check deviation is not that good plus the fact that methodology used to compute price is bad, the impact of this is high

4. The same can be found in BunnySupply.sol `getProtocolOwnedLiquidityReserves()`:

```
function getProtocolOwnedLiquidityReserves()
    external
    view
    override
    returns (SPPLYv1.Reserves[] memory)
{
    // Iterate through tokens and total up the reserves of each pool
    uint256 len = bunniTokens.length;
    SPPLYv1.Reserves[] memory reserves = new SPPLYv1.Reserves[](len);
    for (uint256 i; i < len; ) {
        TokenData storage tokenData = bunniTokens[i];
        BunniToken token = tokenData.token;
        BunniLens lens = tokenData.lens;
        BunniKey memory key = _getBunniKey(token);
        (
            address token0,
            address token1,
            uint256 reserve0,
            uint256 reserve1
        ) = _getReservesWithFees(key, lens);

        // Validate reserves
        _validateReserves(
            key,
            lens,
            tokenData.twapMaxDeviationBps,
            tokenData.twapObservationWindow
        );

        address[] memory underlyingTokens = new address[](2);
        underlyingTokens[0] = token0;
        underlyingTokens[1] = token1;
        uint256[] memory underlyingReserves = new uint256[](2);
        underlyingReserves[0] = reserve0;
        underlyingReserves[1] = reserve1;

        reserves[i] = SPPLYv1.Reserves({
            source: address(token),
            tokens: underlyingTokens,
            balances: underlyingReserves
        });

        unchecked {
            ++i;
        }
    }

    return reserves;
}
```

Where returned value does not account for uncollected fees whereas deviation check was accounting for it

Impact

`_getTotalValue()` from BunniPrice.sol and `getProtocolOwnedLiquidityReserves()` from BunniSupply.sol have both ratio computation that includes uncollected fees to compare with TWAP ratio, potentially overestimating the total value compared to what these functions are aim to, which is returning only the reserves or LP Prices by only taking into account the reserves of the pool. Meaning the check could pass in certain conditions where fees are included in the ratio computation and the deviation check process whereas the deviation check should not have pass without the fees accounted.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/SPPLY/submodules/BunniSupply.sol#L212-L260
https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/PRICE/submodules/feeds/BunniPrice.sol#L110

Tool used

Manual Review

Recommendation

Align the methodology used in both the deviation check and the final price computation. This could involve either including the uncollected fees in both calculations or excluding them in both.

It's ok for BunniSupply as there are 2 functions handling both reserves and reserves+fees but change deviation check process on the second one to include only reserves when checking deviation twap ratio

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

nirohgo commented:

> True observation but the effect on deviation is miniscule and no viable scenario has been shown that leads to a loss of material amounts.

0xJem

Accurate that uncollected fees are excluded from the TWAP check but included in the reserves check, which could lead to inconsistencies. This has been made consistent now.

> this approach of directly computing the price of LP tokens via spot reserves is well-known to be vulnerable to manipulation

We are aware, hence the reserves & TWAP check, plus re-entrancy check.

0xrusowsky

https://github.com/OlympusDAO/bophades/pull/244
https://github.com/OlympusDAO/bophades/pull/246

IAm0x52

Fix looks good. Fees are now included in determining bunni token price. Fees are now not considered in BunniHelper#getFullRangeBunniKey

Issue M-3: Price can be miscalculated.

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/56

Found by

dany.armstrong90

Summary

In `SimplePriceFeedStrategy.sol#getMedianPrice` function, when the length of `nonZeroPrices` is 2 and they are deviated it returns first non-zero value, not median value.

Vulnerability Detail

`SimplePriceFeedStrategy.sol#getMedianPriceIfDeviation` is as follows.

```
    function getMedianPriceIfDeviation(
        uint256[] memory prices_,
        bytes memory params_
    ) public pure returns (uint256) {
        // Misconfiguration
        if (prices_.length < 3) revert SimpleStrategy_PriceCountInvalid(prices_.length, 3);

237     uint256[] memory nonZeroPrices = _getNonZeroArray(prices_);

        // Return 0 if all prices are 0
        if (nonZeroPrices.length == 0) return 0;

        // Cache first non-zero price since the array is sorted in place
        uint256 firstNonZeroPrice = nonZeroPrices[0];

        // If there are not enough non-zero prices to calculate a median, return the first non-zero price
246     if (nonZeroPrices.length < 3) return firstNonZeroPrice;

        uint256[] memory sortedPrices = nonZeroPrices.sort();

        // Get the average and median and abort if there's a problem
        // The following two values are guaranteed to not be 0 since sortedPrices only contains non-zero values and has a length of 3+
        uint256 averagePrice = _getAveragePrice(sortedPrices);
253     uint256 medianPrice = _getMedianPrice(sortedPrices);
```

```
        if (params_.length != DEVIATION_PARAMS_LENGTH) revert
↪   SimpleStrategy_ParamsInvalid(params_);
        uint256 deviationBps = abi.decode(params_, (uint256));
        if (deviationBps <= DEVIATION_MIN || deviationBps >= DEVIATION_MAX)
            revert SimpleStrategy_ParamsInvalid(params_);

        // Check the deviation of the minimum from the average
        uint256 minPrice = sortedPrices[0];
262     if (((averagePrice - minPrice) * 10000) / averagePrice > deviationBps)
↪   return medianPrice;

        // Check the deviation of the maximum from the average
        uint256 maxPrice = sortedPrices[sortedPrices.length - 1];
266     if (((maxPrice - averagePrice) * 10000) / averagePrice > deviationBps)
↪   return medianPrice;

        // Otherwise, return the first non-zero value
        return firstNonZeroPrice;
    }
```

As you can see above, on L237 it gets the list of non-zero prices. If the length of this list is smaller than 3, it assumes that a median price cannot be calculated and returns first non-zero price. This is wrong. If the number of non-zero prices is 2 and they are deviated, it has to return median value. The `_getMedianPrice` function called on L253 is as follows.

```
function _getMedianPrice(uint256[] memory prices_) internal pure returns
↪   (uint256) {
    uint256 pricesLen = prices_.length;

    // If there are an even number of prices, return the average of the two
↪   middle prices
    if (pricesLen % 2 == 0) {
        uint256 middlePrice1 = prices_[pricesLen / 2 - 1];
        uint256 middlePrice2 = prices_[pricesLen / 2];
        return (middlePrice1 + middlePrice2) / 2;
    }

    // Otherwise return the median price
    // Don't need to subtract 1 from pricesLen to get midpoint index
    // since integer division will round down
    return prices_[pricesLen / 2];
}
```

As you can see, the median value can be calculated from two values. This problem exists at `getMedianPrice` function as well.

```
function getMedianPrice(uint256[] memory prices_, bytes memory) public pure
↪   returns (uint256) {
    // Misconfiguration
    if (prices_.length < 3) revert
↪   SimpleStrategy_PriceCountInvalid(prices_.length, 3);

    uint256[] memory nonZeroPrices = _getNonZeroArray(prices_);

    uint256 nonZeroPricesLen = nonZeroPrices.length;
    // Can only calculate a median if there are 3+ non-zero prices
    if (nonZeroPricesLen == 0) return 0;
    if (nonZeroPricesLen < 3) return nonZeroPrices[0];

    // Sort the prices
    uint256[] memory sortedPrices = nonZeroPrices.sort();

    return _getMedianPrice(sortedPrices);
}
```

Impact

When the length of `nonZeroPrices` is 2 and they are deviated, it returns first non-zero value, not median value. It causes wrong calculation error.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus-web3-master/blob/main/bophades/src/modules/PRICE/submodules/strategies/SimplePriceFeedStrategy.sol#L246

Tool used

Manual Review

Recommendation

First, `SimplePriceFeedStrategy.sol#getMedianPriceIfDeviation` function has to be rewritten as follows.

```
    function getMedianPriceIfDeviation(
        uint256[] memory prices_,
        bytes memory params_
    ) public pure returns (uint256) {
        // Misconfiguration
        if (prices_.length < 3) revert
 ↪  SimpleStrategy_PriceCountInvalid(prices_.length, 3);

        uint256[] memory nonZeroPrices = _getNonZeroArray(prices_);

        // Return 0 if all prices are 0
        if (nonZeroPrices.length == 0) return 0;

        // Cache first non-zero price since the array is sorted in place
        uint256 firstNonZeroPrice = nonZeroPrices[0];

        // If there are not enough non-zero prices to calculate a median, return
 ↪  the first non-zero price
-       if (nonZeroPrices.length < 3) return firstNonZeroPrice;
+       if (nonZeroPrices.length < 2) return firstNonZeroPrice;

        ...
    }
```

Second, `SimplePriceFeedStrategy.sol#getMedianPrice` has to be modified as following.

```
    function getMedianPrice(uint256[] memory prices_, bytes memory) public pure
 ↪  returns (uint256) {
        // Misconfiguration
        if (prices_.length < 3) revert
 ↪  SimpleStrategy_PriceCountInvalid(prices_.length, 3);

        uint256[] memory nonZeroPrices = _getNonZeroArray(prices_);

        uint256 nonZeroPricesLen = nonZeroPrices.length;
        // Can only calculate a median if there are 3+ non-zero prices
        if (nonZeroPricesLen == 0) return 0;
-       if (nonZeroPricesLen < 3) return nonZeroPrices[0];
+       if (nonZeroPricesLen < 2) return nonZeroPrices[0];

        // Sort the prices
        uint256[] memory sortedPrices = nonZeroPrices.sort();

        return _getMedianPrice(sortedPrices);
    }
```

Discussion

0xJem

Agree with the highlighted issue, disagree with the proposed solution.

0xJem

https://github.com/OlympusDAO/bophades/pull/282

IAm0x52

Fix looks good. Now falls back to getAveragePriceIfDeviation() instead of returning first.

Issue M-4: Price calculation can be manipulated by intentionally reverting some of price feeds.

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/127

Found by

KupiaSec

Summary

Price calculation module iterates through available price feeds for the requested asset, gather prices of non-revert price feeds and then apply strategy on available prices to calculate final asset price. By abusing this functionality, an attacker can let some price feeds revert to get advantage from any manipulated price feed.

Vulnerability Detail

Here we have some methods that attackers can abuse to intentionally revert price feeds.

1. UniswapV3 price feed UniswapV3Price.sol#L210-214

```
// Get the current price of the lookup token in terms of the quote token
(, int24 currentTick, , , , , bool unlocked) = params.pool.slot0();

// Check for re-entrancy
if (unlocked == false) revert UniswapV3_PoolReentrancy(address(params.pool));
```

In UniswapV3 price feed, it reverts if current state is re-entered. An attacker can intentionally revert this price feed by calling it from UniswapV3's callback methods.

2. Balancer price feed BalancerPoolTokenPrice.sol#L388 BalancerPoolTokenPrice.sol#487 BalancerPoolTokenPrice.sol#599 BalancerPoolTokenPrice.sol#748

```
// Prevent re-entrancy attacks
VaultReentrancyLib.ensureNotInVaultContext(balVault);
```

In BalancerPool price feed, it reverts if current state is re-entered. An attacker can intentionally revert this price feed by calling it in the middle of Balancer action.

3. BunniToken price feed BunniPirce.sol#L155-160

```
_validateReserves(
    _getBunniKey(token),
    lens,
    params.twapMaxDeviationsBps,
    params.twapObservationWindow
);
```

In BunniToken price feed, it validates reserves and reverts if it doesn't satisfy deviation. Since BunniToken uses UniswapV3, this can be intentionally reverted by calling it from UniswapV3's mint callback.

Usually for ERC20 token prices, above 3 price feeds are commonly used combined with Chainlink price feed, and optionally with `averageMovingPrice`. There are another two points to consider here:

1. When average moving price is used, it is appended at the end of the price array. OlympusPrice.v2.sol#L160

```
if (asset.useMovingAverage) prices[numFeeds] = asset.cumulativeObs /
↪   asset.numObservations;
```

2. In price calculation strategy, first non-zero price is used when there are 2 valid prices: `getMedianPriceIfDeviation` - SimplePriceFeedStrategy.sol#L246 `getMedianPrice` - SimplePriceFeedStrategy.sol#L313 For `getAveragePrice` and `getAveragePriceIfDeviation`, it uses average price if it deviates.

Based on the information above, here are potential attack vectors that attackers would try:

1. When Chainlink price feed is manipulated, an attacker can disable all three above price feeds intentionally to get advantage of the price manipulation.
2. When Chainlink price feed is not used for an asset, an attacker can manipulate one of above 3 spot price feeds and disable other ones.

When `averageMovingPrice` is used and average price strategy is applied, the manipulation effect becomes half: $\frac{(P+\Delta X)+(P)}{2} = P + \frac{\Delta X}{2}, P = MarketPrice, \Delta X = ManipulatedAmount$

Impact

Attackers can disable some of price feeds as they want with ease, they can get advantage of one manipulated price feed.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus/blob/9c8df76dc9820b4c6605d2e1e6d87dcfa9e50070/bophades/src/modules/PRICE/OlympusPrice.v2.sol#L132-L184

Tool used

Manual Review

Recommendation

For the cases above that price feeds being intentionally reverted, the price calculation itself also should revert without just ignoring it.

Discussion

nevillehuang

Invalid, if a user purposely revert price feeds, they are only affecting their own usage, not the usage of price feeds for other users transactions.

KupiaSecAdmin

Escalate

Hey @nevillehuang - Yes, exactly you are right. What an attacker can manipulate is a spot price using flashloans, so if an attacker purposely disable other price feeds but only leave manipulated price feed, there happens a vulnerability that an attacker can buy tokens at affected price.

sherlock-admin2

> Escalate
>
> Hey @nevillehuang - Yes, exactly you are right. What an attacker can manipulate is a spot price using flashloans, so if an attacker purposely disable other price feeds but only leave manipulated price feed, there happens a vulnerability that an attacker can buy tokens at affected price.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

@KupiaSecAdmin, All of your scenarios are invalid

1. There is no point for somebody to reenter to explicity cause a revert for using the price feed himself
2. Same reason as 1.
3. There is no point for somebody to cause a deviation to explicity cause a revert for using the price feed himself
4. A user cannot manipulate a chainlink price feed since there are no reserves

This is on top of the fact that price submodules are not intended to be called directly, but via the primary price module mentioned in this comment here

KupiaSecAdmin

@nevillehuang - For example, you can manipulate spot price of Uniswap. To make this work, you need to make other price feeds revert because if they are all enabled, average/median price strategy will be taken and manipulated spot price will not take effect.

nevillehuang

@KupiaSecAdmin you cannot make other feeds revert for other user, only yourself, and your submission certainly doesn't prove that it is possible. Besides, to manipulate spot price in uniswap, you will have to manipulate the reserves, which is a known issue in the contest and out of scope.

KupiaSecAdmin

@nevillehuang - I would like to add some notes and scenarios below that I think might be attack vectors. @0xJem - I would be happy to get some feedbacks from the protocol team regarding the issue.

[Notes]

1. (I believe) This price module will be used in other parts of Olympus protocol to determine fair price of OHM(and other ERC20 tokens) at any time by integrating multiple price feeds and applying a strategy(average or median) to different prices to carry out final fair price.
2. The carried out final price will be used to buy/sell OHM tokens using other collaterals in other modules of Olympus protocol.

[Scenario]

1. Let's assume that an attacker can manipulate a spot price of one price feed, e.g. Uni2, Uni3, Bunni. It can not be guaranteed that all spot price feeds work correctly.
2. As a result, we can assume that the attacker can manipulate OHM price of one price feed to $9(for example by manipulating Bunni).

3. However, multiple price feeds are used to calculate fair OHM price, for example, 3 strategies can be used to determine fair OHM price: Chainlink, Uniswap3, Bunni. Thus assume Chainlink returns $11.1 and Uniswap3 returns $11.05 for OHM price.
4. The price strategy takes median strategy, this means manipulating Bunni price feeds does not take effect on final OHM price determination because the median price of ($9, $11.05, $11.1) is $11.05 which could be accepted as fair OHM price.
5. Now, the attacker can intentionally make Uniswap 3 price feed reverting using re-entrancy.
6. When this happens, the only available price feeds are Chainlink and Bunny which are $9 and $11.1. Median price strategy is applied to these feeds thus returning $10 as OHM price, which is affected and this could result in attacker can buy more OHM tokens than expected.

[Thoughts] Price feeds can revert for any reason by accidents so it would actually make sense using try/catch to ignore reverted price feeds. However, price feeds being reverted because of re-entrancy check can not be considered as accidents because it's intentional and unusual behavior. So I think it's the right behavior to revert price calculation itself as a whole when any price feed is reverted by re-entrancy check.

[Claims] @nevillehuang - You were mentioning that I can not make other feeds revert for other users but only for myself. Yes, that's right. An attacker will let some price feeds revert only for himself(and only within a single transaction, they should work fine in other transactions), and it is to manipulate final fair price of tokens regardless of whatever strategy is taken.

nevillehuang

@KupiaSecAdmin Can you provide a coded PoC for your above scenario? I really don't see how step 5 can occur, given price feeds are utilized in separate transactions? How would one users price feed reverting affect another?

> 5. Now, the attacker can intentionally make Uniswap 3 price feed reverting using re-entrancy.

KupiaSecAdmin

@nevillehuang @0xJem - Here's a PoC that shows how price can be manipulated. You can put this test file in same test directory with `PRICE.v2.t.sol`. https://gist.github.com/KupiaSecAdmin/fc7ef6664b191ab2b758a22ab15bf404

Running test: `forge test --fork-url {{MAINNET_RPC}} --fork-block-number 19041673 --match-test testPriceManipulation -vv`

Result:

```
[PASS] testPriceManipulation() (gas: 2299239)
Logs:
  Before: Chainklink/Olympus 6294108760000000000 6308514491323687440
  After: Chainklink/Olympus 6294108760000000000 29508079057029841191

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.69s

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

[Scenario]

1. It calculates UNI price using mainnet's forked data.
2. It is assumed that Olympus uses UniV2 and UniV3 price feeds for calculating UNI price.
3. The test manipulates UniV2 price and intentionally reverts UniV3 price feed, thus the final price is same as manipulated UniV2 price.

[Focus]

1. Even though test shows price manipulation is done via reserves, but reserve manipulation is not the only way of manipulating price, as Olympus integrates further more price feeds and based on protocols.
2. The main point to show from the issue and PoC is that intentionally reverting some price feeds is dangerous because that can be a cause of price manipulation.

0xJem

@0ighty can you weigh in on the risk of a third-party deliberately triggering the re-entrancy lock in the UniV3 pool?

To me, this represents a misconfiguration of the asset price feeds.

If it was a single price feed (UniV3) only, it would be fine, as the price lookup would fail. It's because there's a UniV2 pool in use that this *could* be susceptible to the price manipulation as you described. However, this feels unlikely because:

- The depth of liquidity on the UNI / WETH UniV2 pool is $4.32m, which feels too low for a UniV3 pool (let alone UniV2!), and so we'd be unlikely to use it.
- For an asset that does not have as much liquidity (e.g. we are following this approach for FXS), we track an internal MA and use that, which ensures that any manipulation is smoothed out.

If we were to have UNI defined as an asset, we would be more likely to do this:

- UNI/ETH Chainlink feed

- UNI/wETH UniV3 pool with TWAP

Given the difficulty of manipulation *both* sources, and the deep liquidity of the UniV3 pool ($31.65m), we'd be confident that it would be resilient enough.

nevillehuang

- UNI wasn't mentioned as an integrated token in the contest details, so wouldn't this be invalid?
- Olympus also has many mitigations in place for TWAP manipulation

Czar102

I think this is a really nice finding if true, kudos for the thought process @KupiaSecAdmin!

Since price manipulation itself is out of scope, but the expectation of using multiple price sources should make the price more difficult to manipulate, and because of the bug, the breakdown value falls drastically. Thus I believe it deserves to be a valid Medium.

I'm not sure about the point above, @0xJem could you explain why would such setup be a misconfiguration? From my understanding, any setup using any of these 3 oracles and any other one will be susceptible to manipulation.

nevillehuang

@Czar102 Some questions:

1. Is there anywhere it was indicated that the above uni pools would be used as price feeds? Given the watson made an assumption:

   > assumed that Olympus uses UniV2 and UniV3 price feeds for calculating UNI price.

2. Isn't the additional data provided by the watson still related to manipulation of reserves and like you said out of scope? To me he still hasn't prove that there is any other cause other than manipulating reserves other than stating a possibility? Would be nice if he can prove this issues above scenario of 1 and 2 (reentrancy triggering affecting price feed of other users?)
3. Dont Olympus use an internal MA to mitigate risk of reserve manipulation?

0xJem

> I'm not sure about the point above, @0xJem could you explain why would such setup be a misconfiguration? From my understanding, any setup using any of these 3 oracles and any other one will be susceptible to manipulation.

- Given the risk of a single price feed reverting (causing the 2nd price feed to be used), we would not use a UniV2 (which doesn't have re-entrancy protection and is much more susceptible to manipulation) pool as the second feed.
- Instead of this UniV3 + UniV3 combination, if we were to configure in PRICE for this asset, we would do a Chainlink feed (e.g. UNI-USD, no idea if it exists) and a UniV3 pool.

Czar102

@nevillehuang I believe the assumption you are mentioning in point 1 is just an example and the different price feeds could be anything, like Uni v3 + Uni v3 – one could manipulate one of these and make the other revert, for example.

Regarding point 2, I don't think the crux here is the manipulation of reserves, they may be just off with respect to each other. The point is that the attacker can selectively decide which sources of information to use, impacting the final price reading. The point of using multiple feeds is to make the price more reliable, and they are being made less reliable if you can make the readings be rejected.

Regarding point 3, I believe you could repetitively make the price pass sanity checks, making it exponentially diverge from the real price.

Regarding @0xJem's points: I believe simply not using a Uni v2 pool doesn't mitigate this. Using any of the dexes mentioned above together with any feed will have this impact. So, a Chainlink feed + Uni v3 pool could be exploited in a way that the Uni v3 reading will revert and only Chainlink feed will be used, which may benefit the attacker in a certain way.

Has the approach for creating these safe setups been shared with Watsons anywhere? Am I misunderstanding something? @0xJem @nevillehuang @KupiaSecAdmin

nevillehuang

@Czar102

- What is the cost of manipulating such price feeds, is it even profitable for the user?
- The ORIGINAL issue certainly doesn't have sufficient proof to prove that anything other than manipulation of reserves will cause price feed revert or show that it is viable/economically viable. Until the watson prove to me with a reasonable PoC that it is possible, I cannot verify validity, especially not with information from the original submission. If a judge has to do alot of additional research apart from what is provided in the issue, it certainly doesn't help too.

  2. In case of non-obvious issues with complex vulnerabilities/attack paths, Watson must submit a valid POC for the issue to be considered valid and rewarded.

- The watson is speculating on how protocol will configure and select different price feeds. Like @0xJem mentioned, this is protocol determined so the above mentioned possibilities are all possible assumptions. “Could be anything” is a weak argument and based off your previous statement here it doesn’t line up, given configurations of price feeds are not explicitly mentioned in docs

TLDR, unless the watson or YOU provide sufficient proof (best with a PoC) that it is economically possible/profitable, I’m not convinced this is a valid issue since you are just simply stating possibility. Please only consider the original submission only and see if it has sufficient information in place during the time when Im judging this.

Hash01011122

IMO In my opinion, while the precise impact of the potential attack isn't crystal clear, the mentioned attack path, extending up to price manipulation, significantly expands the attack surface. This broader surface introduces multiple avenues for potential attacks that may not be immediately apparent. I find @nevillehuang's comment lacking in persuasiveness, on how this issue should be considered as invalid after watson submitted the PoC. With a clear attack impact, Watson's submission should be rated as High severity. Watson's failure to articulate how the identified issue could result in a loss of funds for the protocol is crucial. But the issue highlights numerous ways the core functionality of the contract could be exploited, making it a valid medium-severity concern.

nevillehuang

@Hash01011122, stating the possibility of an issue and proving it are two separate things. Can you look at the details provided in the issue and tell me with at least 80% confidence rate that it is valid without additional research by the judge to prove its validity when its not the case?

For example, the watson is simply stating "user can cause reentrancy" with a single one liner type comment without any code description/POC (there are multiple instances throughout the issue)? How am I suppose to verify that? I am a firm believer that burden of proof is on the watson not the judge, and I believe sherlock also enforces this stance.

The fact that Head of judging and sponsor has to come in and supplement the non-obvious finding of the watson certainly doesn't help too, and I believe this will be resolved in the future now that we have the `request poc` feature, but I believe as of contest date, the information provided in the ORIGINAL submission is insufficient to warrant its severity other than low/invalid.

Czar102

I understood the finding when I haven't read a half or it. I think the only thing that needs to be verified is that a revert in price reading will cause the price to be computed based on other sources.

Selective manipulation of sources of information defeats the purpose of sourcing the data from many sources – instead of increasing security, the data will be pulled from potentially least safe sources.

I think it warrants Medium severity.

nevillehuang

@Czar102 ok got it I put it on myself for not having the knowledge u possess to understand this issue. I will let you decide once you decide what @0xJem considers. Again understanding and proving to issue is two separate issues for debate.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: accepted

Issue M-5: getReservesByCategory() when useSubmodules =true and submoduleReservesSelector=bytes4(0) will revert

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/149

Found by

bin2chen, dany.armstrong90, lemonmon, rvierdiiev

Summary

in `getReservesByCategory()` Lack of check `data.submoduleReservesSelector!=""` when call `submodule.staticcall(abi.encodeWithSelector(data.submoduleReservesSelector));` will revert

Vulnerability Detail

when `_addCategory()` if `useSubmodules==true`, `submoduleMetricSelector` must not empty and `submoduleReservesSelector` can empty (bytes4(0))

like "protocol-owned-treasury"

```
_addCategory(toCategory("protocol-owned-treasury"), true, 0xb600c5e2,
↪   0x00000000); // getProtocolOwnedTreasuryOhm()`
```

but when call `getReservesByCategory()` , don't check `submoduleReservesSelector!=bytes4(0)` and direct call `submoduleReservesSelector`

```
    function getReservesByCategory(
        Category category_
    ) external view override returns (Reserves[] memory) {
...
        // If category requires data from submodules, count all submodules and
↪   their sources.
        len = (data.useSubmodules) ? submodules.length : 0;

...

        for (uint256 i; i < len; ) {
            address submodule = address(_getSubmoduleIfInstalled(submodules[i]));
            (bool success, bytes memory returnData) = submodule.staticcall(
                abi.encodeWithSelector(data.submoduleReservesSelector)
            );
```

this way , when call like `getReservesByCategory(toCategory("protocol-owned-treasury")` will revert

POC

add to `SUPPLY.v1.t.sol`

```
function test_getReservesByCategory_includesSubmodules_treasury() public {
    _setUpSubmodules();

    // Add OHM/gOHM in the treasury (which will not be included)
    ohm.mint(address(treasuryAddress), 100e9);
    gohm.mint(address(treasuryAddress), 1e18); // 1 gOHM

    // Categories already defined

    uint256 expectedBptDai = BPT_BALANCE.mulDiv(
        BALANCER_POOL_DAI_BALANCE,
        BALANCER_POOL_TOTAL_SUPPLY
    );
    uint256 expectedBptOhm = BPT_BALANCE.mulDiv(
        BALANCER_POOL_OHM_BALANCE,
        BALANCER_POOL_TOTAL_SUPPLY
    );

    // Check reserves
    SPPLYv1.Reserves[] memory reserves = moduleSupply.getReservesByCategory(
        toCategory("protocol-owned-treasury")
    );
}
```

```
 forge test -vv --match-test
↪   test_getReservesByCategory_includesSubmodules_treasury

Running 1 test for src/test/modules/SPPLY/SPPLY.v1.t.sol:SupplyTest
[FAIL. Reason: SPPLY_SubmoduleFailed(0xeb502B1d35e975321B21cCE0E8890d20a7Eb289d,
↪   0x0000000000000000000000000000000000000000000000000000000000000000)]
↪   test_getReservesByCategory_includesSubmodules_treasury() (gas: 4774197
```

Impact

some category can't get `Reserves`

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/SPPLY/OlympusSupply.sol#L541

Tool used

Manual Review

Recommendation

```
    function getReservesByCategory(
        Category category_
    ) external view override returns (Reserves[] memory) {
...


        CategoryData memory data = categoryData[category_];
        uint256 categorySubmodSources;
        // If category requires data from submodules, count all submodules and
 ↪  their sources.
-       len = (data.useSubmodules) ? submodules.length : 0;
+       len = (data.useSubmodules && data.submoduleReservesSelector!=bytes4(0))
 ↪  ? submodules.length : 0;
```

Discussion

0xJem

Good catch! Thank you for the clear explanation and test case, too.

0xrusowsky

https://github.com/OlympusDAO/bophades/pull/262

IAm0x52

Fix looks good, exactly as suggested

Issue M-6: Balancer LP valuation methodologies use the incorrect supply metric

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/155

Found by

0x52, 0xMR0, Arabadzhiev, AuditorPraise, Bauer, CL001, Drynooo, ZanyBonzy, ast3ros, bin2chen, coffiasd, cu5t0mPe0, ge6a, hash, jovi, shealtielanz, tvdung94

Summary

In various Balancer LP valuations, totalSupply() is used to determine the total LP supply. However this is not the appropriate method for determining the supply. Instead getActualSupply should be used instead. Depending on the which pool implementation and how much LP is deployed, the valuation can be much too high or too low. Since the RBS pricing is dependent on this metric. It could lead to RBS being deployed at incorrect prices.

Vulnerability Detail

AuraBalancerSupply.sol#L345-L362

```
uint256 balTotalSupply = pool.balancerPool.totalSupply();
uint256[] memory balances = new uint256[](_vaultTokens.length);
// Calculate the proportion of the pool balances owned by the polManager
if (balTotalSupply != 0) {
    // Calculate the amount of OHM in the pool owned by the polManager
    // We have to iterate through the tokens array to find the index of OHM
    uint256 tokenLen = _vaultTokens.length;
    for (uint256 i; i < tokenLen; ) {
        uint256 balance = _vaultBalances[i];
        uint256 polBalance = (balance * balBalance) / balTotalSupply;


        balances[i] = polBalance;


        unchecked {
            ++i;
        }
    }
}
```

To value each LP token the contract divides the valuation of the pool by the total supply of LP. This in itself is correct, however the totalSupply method for a variety of Balancer pools doesn't accurately reflect the true LP supply. If we take a look at a few Balancer pools we can quickly see the issue:

This pool shows a max supply of 2,596,148,429,273,858 whereas the actual supply is 6454.48. In this case the LP token would be significantly undervalued. If a sizable portion of the reserves are deployed in an affected pool the backing per OHM would appear to the RBS system to be much lower than it really is. As a result it can cause the RBS to deploy its funding incorrectly, potentially selling/buying at a large loss to the protocol.

Impact

Pool LP can be grossly under/over valued

Code Snippet

AuraBalancerSupply.sol#L332-L369

Tool used

Manual Review

Recommendation

Use a try-catch block to always query getActualSupply on each pool to make sure supported pools use the correct metric.

Discussion

0xJem

This is a valid issue and highlights problems with Balancer's documentation.

We are likely to drop both the Balancer submodules from the final version, since we no longer have any Balancer pools used for POL and don't have any assets that require price resolution via Balancer pools.

Issue M-7: Possible incorrect price for tokens in Balancer stable pool due to amplification parameter update

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/178

Found by

hash

Summary

Incorrect price calculation of tokens in StablePools if amplification factor is being updated

Vulnerability Detail

The amplification parameter used to calculate the invariant can be in a state of update. In such a case, the current amplification parameter can differ from the amplificaiton parameter at the time of the last invariant calculation. The current implementaiton of `getTokenPriceFromStablePool` doesn't consider this and always uses the amplification factor obtained by calling `getLastInvariant`

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/PRICE/submodules/feeds/BalancerPoolTokenPrice.sol#L811-L827

```
        function getTokenPriceFromStablePool(
    address lookupToken_,
    uint8 outputDecimals_,
    bytes calldata params_
) external view returns (uint256) {

            .....

            try pool.getLastInvariant() returns (uint256, uint256 ampFactor) {

               // @audit the amplification factor as of the last invariant
↪   calculation is used
                lookupTokensPerDestinationToken = StableMath._calcOutGivenIn(
                    ampFactor,
                    balances_,
                    destinationTokenIndex,
                    lookupTokenIndex,
                    1e18,
                    StableMath._calculateInvariant(ampFactor, balances_) //
↪   Sometimes the fetched invariant value does not work, so calculate it
```

```
                 );
```

https://vscode.blockscan.com/ethereum/0x1e19cf2d73a72ef1332c882f20534b6519be0276 StablePool.sol

```
    // @audit the amplification parameter can be updated
    function startAmplificationParameterUpdate(uint256 rawEndValue, uint256
↪   endTime) external authenticate {

    // @audit for calculating the invariant the current amplification factor is
↪   obtained by calling _getAmplificationParameter()
    function _onSwapGivenIn(
    SwapRequest memory swapRequest,
    uint256[] memory balances,
    uint256 indexIn,
    uint256 indexOut
) internal virtual override whenNotPaused returns (uint256) {
    (uint256 currentAmp, ) = _getAmplificationParameter();
    uint256 amountOut = StableMath._calcOutGivenIn(currentAmp, balances,
↪   indexIn, indexOut, swapRequest.amount);
    return amountOut;
}
```

Impact

In case the amplification parameter of a pool is being updated by the admin, wrong price will be calculated.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/PRICE/submodules/feeds/BalancerPoolTokenPrice.sol#L811-L827

Tool used

Manual Review

Recommendation

Use the latest amplification factor by callling the `getAmplificationParameter` function

Discussion

0xJem

This doesn't seem valid - if the amplification factor is changed since the invariant was last calculated, wouldn't the value of the invariant also be invalid?

nevillehuang

Hi @0xJem here is additional information provided by watson:

The invariant used for calculating swap amounts in Balancer is always based on the latest amplification factor hence their calculation would be latest. If there are no join actions, the cached amplification factor which is used by Olympus will not reflect the new one and will result in a different invariant and different token price.

i am attaching a poc if required: https://gist.github.com/10xhash/8e24d0765ee98def8c6409c71a7d2b17

0xauditsea

Escalate

This looks like invalid. Logically thinking, using `getLastInvariant` is more precise because the goal of this price feed is to calculate spot price of the balancer pool. If current amplification factor is used, it doesn't represent current state of the pool.

sherlock-admin2

> Escalate
>
> This looks like invalid. Logically thinking, using `getLastInvariant` is more precise because the goal of this price feed is to calculate spot price of the balancer pool. If current amplification factor is used, it doesn't represent current state of the pool.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Czar102

@nevillehuang what do you think?

nevillehuang

@Czar102 I don't quite understand what @0xauditsea is pointing to. If you want to calculate the latest spot price, shouldn't you use the latest factor as indicated by the PoC by @10xhash?

Czar102

@0xauditsea could you explain your reasoning in more detail?

nevillehuang

@10xhash does this affect ALL pools intended to be integrated during the time of contest?

10xhash

It has to be clarified what `intended to be integrated pools` at the time of contest are:

1. If only the list of tokens mentioned in the readme *can be* in a pool (as mentioned in previous replies this is not required per the contest definition since all tokens are not required to interact with contracts) : There are 0 stable pools including normal, metastable etc. The only possible stable pool of any type that can be used with the above restriction is the dai-sdai metastable pool which has to be deployed in future.
2. Else it must atleast include normal stable pools and according to balancer's documentation {search startAmplificationParameterUpdate} and testing done on the dai-usdc-usdt pool, it would be affected

Czar102

1. Metastable pools are not supposed to be supported.
2. This documentation seems to be for Avalanche, while the contracts will be deployed on mainnet. I believe this functionality exists on mainnet too, right?

Aside from that, the impact is that the price calculated is the price at the last pool update (trade) instead of the current price?

10xhash

2. The link opens up to Mainnet for me, if not you would have the option to select the chain on leftside. Yes.

The impact would be that the amplification parameter used in the price calculation will be that of the last join action (addliquidity , removeliquidity) which will be different from the actual one used in the pool calculations. This will result in an incorrect price until some user performs a join operation.

Czar102

Adding/removing liquidity doesn't necessarily happen often. This, together with the amplification parameter change, is a very unlikely situation, nevertheless a possible one.

It's a borderline Med/Low, but I am inclined to keep this one a valid Medium. I don't understand the point made in the escalation, and @0xauditsea hasn't elaborated when asked for additional information.

gstoyanovbg

In determining the impact of this report, in my opinion, it should be assessed how much the price can change in the described circumstances and whether the change is significant. I conducted a foundry test that shows the change in the price of AURA_BAL at different values of the amplification factor. The test should be added to BalancerPoolTokenPriceStable.t.sol.

```
function test_amp_factor_impact() public {
        bytes memory params = encodeBalancerPoolParams(mockStablePool);
        uint256 price;

        mockStablePool.setLastInvariant(INVARIANT, AMP_FACTOR);
        price = balancerSubmodule.getTokenPriceFromStablePool(
            AURA_BAL,
            PRICE_DECIMALS,
            params
        );
        console.log("%d, AMP_FACTOR = 50000", price);

        mockStablePool.setLastInvariant(INVARIANT, AMP_FACTOR + 2000);
        price = balancerSubmodule.getTokenPriceFromStablePool(
            AURA_BAL,
            PRICE_DECIMALS,
            params
        );
        console.log("%d, AMP_FACTOR = 50000 + 2000", price);

        mockStablePool.setLastInvariant(INVARIANT, AMP_FACTOR + 10000);
        price = balancerSubmodule.getTokenPriceFromStablePool(
            AURA_BAL,
            PRICE_DECIMALS,
            params
        );
        console.log("%d, AMP_FACTOR = 50000 + 10000", price);

        mockStablePool.setLastInvariant(INVARIANT, AMP_FACTOR * 2);
        price = balancerSubmodule.getTokenPriceFromStablePool(
            AURA_BAL,
            PRICE_DECIMALS,
            params
        );
        console.log("%d, AMP_FACTOR = 50000 * 2", price);

        mockStablePool.setLastInvariant(INVARIANT, AMP_FACTOR * 4);
        price = balancerSubmodule.getTokenPriceFromStablePool(
            AURA_BAL,
            PRICE_DECIMALS,
            params
        );
        console.log("%d, AMP_FACTOR = 50000 * 4", price);

        mockStablePool.setLastInvariant(INVARIANT, AMP_FACTOR * 10);
        price = balancerSubmodule.getTokenPriceFromStablePool(
            AURA_BAL,
            PRICE_DECIMALS,
            params
        );
        console.log("%d, AMP_FACTOR = 50000 * 10", price);

        mockStablePool.setLastInvariant(INVARIANT, AMP_FACTOR * 100);
        price = balancerSubmodule.getTokenPriceFromStablePool(
            AURA_BAL,
            PRICE_DECIMALS,
            params
        );
        console.log("%d, AMP_FACTOR = 50000 * 100", price);
    }
```

```
16602528871962134544, AMP_FACTOR = 50000
16606565178508667081, AMP_FACTOR = 50000 + 2000
16620074517406602667, AMP_FACTOR = 50000 + 10000
16655599693391809126, AMP_FACTOR = 50000 * 2
16682630482761745824, AMP_FACTOR = 50000 * 4
16699011129392628938, AMP_FACTOR = 50000 * 10
16708898633935285195, AMP_FACTOR = 50000 * 100
```

From the obtained results, it can be seen that the change in price is small. Even if we increase it 100 times to the maximum possible value of 5000 * 10ˆ3, the change in price is around 0.1 (0.63%). For such a large increase of the amplification factor, it would take about 7 days (2x per day). Another question is what is the chance that there will be no join or exit within these 7 days.

@Czar102 I don't know if this is significant enough change in the price for Sherlock, but wanted to share it to be sure it will be taken into consideration.

Czar102

@gstoyanovbg Thank you for the test, it looks like this should be a low severity issue.

@10xhash Can you provide a scenario where the price would be altered by more than 5%?

10xhash

Place the test inside test/ and run `forge test --mt testHash_AmplificationDiff5` It is asserted that the diff in price is > 5% when the current amplification parameter is divided by 6 with a 4 day period. Dividing by 6 would make the pool close to 8000 (currently 50000).

```
pragma solidity 0.8.15;

import "forge-std/Test.sol";
import {IStablePool} from "src/libraries/Balancer/interfaces/IStablePool.sol";
import {IVault} from "src/libraries/Balancer/interfaces/IVault.sol";
import {FullMath} from "src/libraries/FullMath.sol";
import {StableMath} from "src/libraries/Balancer/math/StableMath.sol";
import {IVault} from "src/libraries/Balancer/interfaces/IVault.sol";
import {IBasePool} from "src/libraries/Balancer/interfaces/IBasePool.sol";
import {IWeightedPool} from
↪   "src/libraries/Balancer/interfaces/IWeightedPool.sol";
import {IStablePool} from "src/libraries/Balancer/interfaces/IStablePool.sol";
import {VaultReentrancyLib} from
↪   "src/libraries/Balancer/contracts/VaultReentrancyLib.sol";
import {LogExpMath} from "src/libraries/Balancer/math/LogExpMath.sol";
import {FixedPoint} from "src/libraries/Balancer/math/FixedPoint.sol";

interface IStablePoolWithAmp is IStablePool {
    function getAmplificationParameter()
        external
        view
        returns (uint amp, bool isUpdating, uint precision);



        function startAmplificationParameterUpdate(uint256 rawEndValue, uint256
↪   endTime) external;
}

interface IERC20 {
    function approve(address spender,uint amount) external;
}

enum SwapKind { GIVEN_IN, GIVEN_OUT }


struct SingleSwap {
        bytes32 poolId;
        SwapKind kind;
        address assetIn;
        address assetOut;
        uint256 amount;
        bytes userData;
    }

        struct FundManagement {
        address sender;
        bool fromInternalBalance;
        address payable recipient;
        bool toInternalBalance;
    }

interface VaultWithSwap is IVault{
    function swap(
        SingleSwap memory singleSwap,
        FundManagement memory funds,
        uint256 limit,
        uint256 deadline
    ) external payable returns (uint256);
}

contract PriceTest is Test {
    using FullMath for uint256;

    function testHash_AmplificationDiff5() public {
        VaultWithSwap balVault =
↪   VaultWithSwap(0xBA12222222228d8Ba445958a75a0704d566BF2C8);
        IStablePoolWithAmp pool =
↪   IStablePoolWithAmp(0x3dd0843A028C86e0b760b1A76929d1C5Ef93a2dd);

        (, uint cachedAmpFactor) = pool.getLastInvariant();
        {
            (, bool isUpdating, ) = pool.getAmplificationParameter();
             assert(isUpdating == false);
        }

        console.log("cahced factor",cachedAmpFactor);

        {
        address mainnetFeeSetter = 0xf4A80929163C5179Ca042E1B292F5EFBBE3D89e6;

        vm.startPrank(mainnetFeeSetter);
        pool.startAmplificationParameterUpdate(cachedAmpFactor / 6 / 1e3,
↪   block.timestamp + 4 days);

        vm.warp(block.timestamp + 4 days + 100);

        // perform swaps to update the balances with latest amp factor
        {
            (uint amp,bool isUpdating , ) = pool.getAmplificationParameter();
            assert(isUpdating == false);
        }

        console.log("amp params set");
        }

        uint[] memory balances_;
        uint actualAmpFactor;
{
bytes32 poolId = pool.getPoolId();
        (actualAmpFactor, , ) = pool.getAmplificationParameter();

        (,  balances_, ) = balVault.getPoolTokens(poolId);
        uint256[] memory scalingFactors = pool.getScalingFactors();
        {

            uint256 len = scalingFactors.length;
            for (uint256 i; i < len; ++i) {
                balances_[i] = FixedPoint.mulDown(balances_[i],
 ↪  scalingFactors[i]);
            }
        }
}


        // lookup token auraBal and destination token lp token

        uint oldCachedPrice;
        uint newAmpFactorPrice;
        {
            uint destinationTokenIndex = 0;
        uint lookupTokenIndex = 1;
            console.log("calculation with previous amp factor");
            uint lookupTokensPerDestinationToken;
            lookupTokensPerDestinationToken = StableMath._calcOutGivenIn(
                cachedAmpFactor,
                balances_,
                destinationTokenIndex,
                lookupTokenIndex,
                1e18,
                StableMath._calculateInvariant(cachedAmpFactor, balances_)
            );

            // Downscale the amount to token decimals
        uint256[] memory scalingFactors = pool.getScalingFactors();
            lookupTokensPerDestinationToken = FixedPoint.divDown(
                lookupTokensPerDestinationToken,
                scalingFactors[lookupTokenIndex]
            );

            uint outputDecimals = 8;

            lookupTokensPerDestinationToken =
                (lookupTokensPerDestinationToken * 10 ** outputDecimals) /
                1e18;

            uint destinationTokenPrice =  1127000000;
            console.log("bal lp price", destinationTokenPrice);
            uint lookupTokenPrice;

            lookupTokenPrice = destinationTokenPrice.mulDiv(
                10 ** outputDecimals,
                lookupTokensPerDestinationToken
            );
            oldCachedPrice = lookupTokenPrice;
            console.log("aurabal price", lookupTokenPrice);
        }

        {
            uint destinationTokenIndex = 0;
        uint lookupTokenIndex = 1;
            console.log("calculation with updated amp factor");
            uint lookupTokensPerDestinationToken;
            lookupTokensPerDestinationToken = StableMath._calcOutGivenIn(
                actualAmpFactor,
                balances_,
                destinationTokenIndex,
                lookupTokenIndex,
                1e18,
                StableMath._calculateInvariant(actualAmpFactor, balances_)
            );

            // Downscale the amount to token decimals
        uint256[] memory scalingFactors = pool.getScalingFactors();
            lookupTokensPerDestinationToken = FixedPoint.divDown(
                lookupTokensPerDestinationToken,
                scalingFactors[lookupTokenIndex]
            );

            uint outputDecimals = 8;

            lookupTokensPerDestinationToken =
                (lookupTokensPerDestinationToken * 10 ** outputDecimals) /
```

```
                1e18;

            uint destinationTokenPrice = 1127000000;
            console.log("bal lp price", destinationTokenPrice);
            uint lookupTokenPrice;

            lookupTokenPrice = destinationTokenPrice.mulDiv(
                10 ** outputDecimals,
                lookupTokensPerDestinationToken
            );
            newAmpFactorPrice = lookupTokenPrice;
            console.log("aurabal price", lookupTokenPrice);
        }

        assert((oldCachedPrice -newAmpFactorPrice) * 100 * 1e18 /
↪   newAmpFactorPrice > 5 ether);

    }

}
```

gstoyanovbg

@10xhash well done, i think your test is valid and shows a significant price change.

Czar102

Thank you @10xhash! Planning to leave the issue as is.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0xauditsea: rejected

Issue M-8: Incorrect deviation calculation in isDeviatingWithBpsCheck function

Source: https://github.com/sherlock-audit/2023-11-olympus-judging/issues/193

Found by

ast3ros, coffiasd, dany.armstrong90, evilakela, hash

Summary

The current implementation of the `isDeviatingWithBpsCheck` function in the codebase leads to inaccurate deviation calculations, potentially allowing deviations beyond the specified limits.

Vulnerability Detail

The function `isDeviatingWithBpsCheck` checks if the deviation between two values exceeds a defined threshold. This function incorrectly calculates the deviation, considering only the deviation from the larger value to the smaller one, instead of the deviation from the mean (or TWAP).

```
function isDeviatingWithBpsCheck(
    uint256 value0_,
    uint256 value1_,
    uint256 deviationBps_,
    uint256 deviationMax_
) internal pure returns (bool) {
    if (deviationBps_ > deviationMax_)
        revert Deviation_InvalidDeviationBps(deviationBps_, deviationMax_);

    return isDeviating(value0_, value1_, deviationBps_, deviationMax_);
}

function isDeviating(
    uint256 value0_,
    uint256 value1_,
    uint256 deviationBps_,
    uint256 deviationMax_
) internal pure returns (bool) {
    return
        (value0_ < value1_)
            ? _isDeviating(value1_, value0_, deviationBps_, deviationMax_)
            : _isDeviating(value0_, value1_, deviationBps_, deviationMax_);
}
```

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/libraries/Deviation.sol#L23-L52

The function then call `_isDeviating` to calculate how much the smaller value is deviated from the bigger value.

```
function _isDeviating(
    uint256 value0_,
    uint256 value1_,
    uint256 deviationBps_,
    uint256 deviationMax_
) internal pure returns (bool) {
    return ((value0_ - value1_) * deviationMax_) / value0_ > deviationBps_;
}
```

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/libraries/Deviation.sol#L63-L70

The function `isDeviatingWithBpsCheck` is usually used to check how much the current value is deviated from the TWAP value to make sure that the value is not manipulated. Such as spot price and twap price in UniswapV3.

```
if (
    // `isDeviatingWithBpsCheck()` will revert if `deviationBps` is invalid.
    Deviation.isDeviatingWithBpsCheck(
        baseInQuotePrice,
        baseInQuoteTWAP,
        params.maxDeviationBps,
        DEVIATION_BASE
    )
) {
    revert UniswapV3_PriceMismatch(address(params.pool), baseInQuoteTWAP,
     ↪  baseInQuotePrice);
}
```

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/modules/PRICE/submodules/feeds/UniswapV3Price.sol#L225-L235

The issue is isDeviatingWithBpsCheck is not check the deviation of current value to the TWAP but deviation from the bigger value to the smaller value. This leads to an incorrect allowance range for the price, permitting deviations that exceed the acceptable threshold.

Example:

TWAP price: 1000 Allow deviation: 10%.

The correct deviation calculation will use deviation from the mean. The allow price will be from 900 to 1100 since:

- |1100 - 1000| / 1000 = 10%
- |900 - 1000| / 1000 = 10%

However the current calculation will allow the price from 900 to 1111

- (1111 - 1000) / 1111 = 10%
- (1000 - 900) / 1000 = 10%

Even though the actual deviation of 1111 to 1000 is |1111 - 1000| / 1000 = 11.11% > 10%

Impact

This miscalculation allows for greater deviations than intended, increasing the vulnerability to price manipulation and inaccuracies in Oracle price reporting.

Code Snippet

https://github.com/sherlock-audit/2023-11-olympus/blob/main/bophades/src/libraries/Deviation.sol#L63-L70

Tool used

Manual review

Recommendation

To accurately measure deviation, the isDeviating function should be revised to calculate the deviation based on the mean value: `| spot value - twap value | / twap value`.

Discussion

0xrusowsky

https://github.com/OlympusDAO/bophades/pull/245

IAm0x52

Escalate

This is purely a design choice. Nothing here is wrong with the implementation. The deviation is purely subjective and is measured objectively the same in both directions. This should be a low severity issue in my opinion and I strongly believe it should be. At the maximum this should be a medium severity issues as impact is not large at all for any reasonable variation and only subjectively incorrect

sherlock-admin2

> Escalate
>
> This is purely a design choice. Nothing here is wrong with the implementation. The deviation is purely subjective and is measured objectively the same in both directions. This should be a low severity issue in my opinion and I strongly believe it should be. At the maximum this should be a medium severity issues as impact is not large at all for any reasonable variation and only subjectively incorrect

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

@IAm0x52 I'm pretty sure sponsor acknowledging this with a fix indicates this is not a design choice. Let me know if there are any publicly available information at time of contests that points to that I am missing.

Since price values are CORE components of price modules, I labelled it as high as the returned price should never be allowed to have too significant of a deviation if not every use case of this prices will be impacted. I think #3 highlights the possible impact of this issues well, and as such this issues should have a minimum of medium severity if not high.

IAm0x52

This is only used in the BUNNI library which is full range liquidity. This simply used to ensure that reserves have not been manipulated and is not the price being used. Using the example provided at a 10% deviation. Reserves can be ~1% different between methodologies.

Let's do a small bit of math to figure this. Assume current invariant is 10000 and there should be 100 of each token (100 * 100 = 10000). If each token is worth $1 then the true value of the pool is 200 (1 * 100 + 1 * 100) Assume price has been manipulated up 10% so now the pool has 110 and 90.9 (10000 / 110) so the value of the pool is now 200.9 (110 * 1 + 90.9 * 1). Lets move it 1.111% more to 11.111% this means there is 111.1111 and 90 (10000 / 111.111) so the value of the pool is now 201.111 (111.111 * 1 + 90 * 1). This results in a difference of 0.211 on a value of 200.9 or 0.1%. This is entirely negligible and hence why I say the deviation check order is a design choice and either way is negligible.

nevillehuang

@IAm0x52 Agree with your analysis, but on context that core contract functionality of deviation check is broken, suggest to keep medium severity.

IAm0x52

Fix looks good. Benchmark is now always the middle for comparison

Czar102

I agree that calculating deviation in log is a valid design choice. Nevertheless, I think it was clear from the comments in code that the deviation was supposed to be calculated symmetrically and linearly, I acknowledge the limitations of this bug as well.

Hence, planning to consider this a medium severity issue.

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- IAm0x52: accepted

Disclaimers

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

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