SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Notional
Prepared by: Sherlock
Lead Security Expert: xiaoming90
Dates Audited: January 2 - January 9, 2023
Prepared on: January 23, 2023

Introduction

Notional is a protocol on Ethereum that facilitates fixed-rate, fixed-term crypto asset lending and borrowing through a novel financial instrument called fcash.

Scope

PR #35 - Implementing new strategy token valuation model + balancer oracle removal

https://github.com/notional-finance/leveraged-vaults/pull/35/files

Background: As a result of findings from the first Sherlock Leveraged Vault audit, we decided to refactor the oracle valuation method to depend on Chainlink oracles instead of the native Balancer pool TWAP oracle. These changes were too large for the fix review and therefore are included in this audit.

Specific Focus Areas for PR #35:

- Adding AccessControlUpgradeable in contracts/vaults/BaseStrategyVault.sol
- Changes to oracle valuation method:
  - contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol
  - contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol

PR #39 - Fixing boosted pool

https://github.com/notional-finance/leveraged-vaults/pull/39/files

Background: The first Sherlock audit covered the Boosted3TokenAura strategy, however, Balancer has since migrated the Boosted3Token pool from the legacy BoostedPool structure to a new ComposableBoostedPool contract. The changes in this PR are related to adapting to the new ComposableBoostedPool as well as changing the valuation methodology to rely on Chainlink. The relevant Balancer ComposableBoostedPool contract can be found here: https://etherscan.io/address/0xa13a9247ea42d743238089903570127dda72fe44#code

Specific Focus Areas for PR #39:

- Changes to the Boosted3TokenAura Strategy:
  - contracts/vaults/balancer/external/Boosted3TokenAuraHelper.sol
  - contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol
  - contracts/vaults/balancer/internal/math/LinearMath.sol
  - contracts/vaults/balancer/internal/math/StableMath.sol
  - contracts/vaults/balancer/mixins/Boosted3TokenPoolMixin.sol

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
3	9

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

xiaoming90
Jeiwan
thekmj
ak1
MalfurionWhitehat
Deivitto

Issue H-1: Two token vault will be broken if it comprises tokens with different decimals

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/18

Found by

xiaoming90

Summary

A two token vault that comprises tokens with different decimals will have many of its key functions broken. For instance, rewards cannot be reinvested and vault cannot be settled.

Vulnerability Detail

The `Stable2TokenOracleMath._getSpotPrice` function is used to compute the spot price of two tokens.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L15

```
File: Stable2TokenOracleMath.sol
11: library Stable2TokenOracleMath {
12:     using TypeConvert for int256;
13:     using Stable2TokenOracleMath for StableOracleContext;
14:
15:     function _getSpotPrice(
16:         StableOracleContext memory oracleContext,
17:         TwoTokenPoolContext memory poolContext,
18:         uint256 primaryBalance,
19:         uint256 secondaryBalance,
20:         uint256 tokenIndex
21:     ) internal view returns (uint256 spotPrice) {
22:         require(tokenIndex < 2); /// @dev invalid token index
23:
24:         /// Apply scale factors
25:         uint256 scaledPrimaryBalance = primaryBalance *
↪  poolContext.primaryScaleFactor
26:             / BalancerConstants.BALANCER_PRECISION;
27:         uint256 scaledSecondaryBalance = secondaryBalance *
↪  poolContext.secondaryScaleFactor
28:             / BalancerConstants.BALANCER_PRECISION;
29:
```

```
30:         /// @notice poolContext balances are always in BALANCER_PRECISION
 ↪  (1e18)
31:         (uint256 balanceX, uint256 balanceY) = tokenIndex == 0 ?
32:             (scaledPrimaryBalance, scaledSecondaryBalance) :
33:             (scaledSecondaryBalance, scaledPrimaryBalance);
34:
35:         uint256 invariant = StableMath._calculateInvariant(
36:             oracleContext.ampParam, StableMath._balances(balanceX,
 ↪  balanceY), true // round up
37:         );
38:
39:         spotPrice = StableMath._calcSpotPrice({
40:             amplificationParameter: oracleContext.ampParam,
41:             invariant: invariant,
42:             balanceX: balanceX,
43:             balanceY: balanceY
44:         });
45:
46:         /// Apply secondary scale factor in reverse
47:         uint256 scaleFactor = tokenIndex == 0 ?
48:             poolContext.secondaryScaleFactor *
 ↪  BalancerConstants.BALANCER_PRECISION / poolContext.primaryScaleFactor :
49:             poolContext.primaryScaleFactor *
 ↪  BalancerConstants.BALANCER_PRECISION / poolContext.secondaryScaleFactor;
50:         spotPrice = spotPrice * BalancerConstants.BALANCER_PRECISION /
 ↪  scaleFactor;
51:     }
```

Two tokens (USDC and DAI) with different decimals will be used below to illustrate the issue:

USDC/DAI Spot Price Assume that the primary token is DAI (18 decimals) and the secondary token is USDC (6 decimals). As such, the scaling factors would be as follows. The token rate is ignored and set to 1 for simplicity.

- Primary Token (DAI)'s scaling factor = 1e18

```
scaling factor = FixedPoint.ONE (1e18) * decimals difference to reach 18
 ↪  decimals (1e0) * token rate (1)
scaling factor = 1e18
```

- Secondary Token (USDC)'s scaling factor = 1e30

```
scaling factor = FixedPoint.ONE (1e18) * decimals difference to reach 18
 ↪  decimals (1e12) * token rate (1)
scaling factor = 1e18 * 1e12 = 1e30
```

Assume that the `primaryBalance` is 100 DAI (100e18), and the `secondaryBalance` is 100 USDC (100e6). Line 25 - 28 of the `_getSpotPrice` function will normalize the tokens balances to 18 decimals as follows:

- `scaledPrimaryBalance` will be 100e18 (It remains the same as no scaling is needed because DAI is already denominated in 18 decimals)

```
scaledPrimaryBalance = primaryBalance * poolContext.primaryScaleFactor /
↪  BalancerConstants.BALANCER_PRECISION;
scaledPrimaryBalance = 100e18 * 1e18 / 1e18
scaledPrimaryBalance = 100e18
```

- `scaledSecondaryBalance` will upscale to 100e18

```
scaledSecondaryBalance = scaledSecondaryBalance *
↪  poolContext.primaryScaleFactor / BalancerConstants.BALANCER_PRECISION;
scaledSecondaryBalance = 100e6 * 1e30 / 1e18
scaledSecondaryBalance = 100e18
```

The `StableMath._calcSpotPrice` function at Line 39 returns the spot price of Y/X. In this example, `balanceX` is DAI, and `balanceY` is USDC. Thus, the spot price will be USDC/DAI. This means the amount of USDC I will get for each DAI.

Within Balancer, all stable math calculations within the Balancer's pools are performed in 1e18. With both the primary and secondary balances normalized to 18 decimals, they can be safely passed to the `StableMath._calculateInvariant` and `StableMath._calcSpotPrice` functions to compute the spot price. Assuming that the price of USDC and DAI is perfectly symmetric (1 DAI can be exchanged for exactly 1 USDC, and vice versa), the spot price returned from the `StableMath._calcSpotPrice` will be `1e18`. Note that the spot price returned by the `StableMath._calcSpotPrice` function will be denominated in 18 decimals.

In Line 47-50 within the `Stable2TokenOracleMath._getSpotPrice` function, it attempts to downscale the spot price to normalize it back to the original decimals and token rate (e.g. stETH back to wstETH) of the token.

The `scaleFactor` at Line 47 will be evaluated as follows:

```
scaleFactor = poolContext.secondaryScaleFactor *
↪  BalancerConstants.BALANCER_PRECISION / poolContext.primaryScaleFactor
scaleFactor = 1e30 * 1e18 / 1e18
scaleFactor = 1e30
```

Finally, the spot price will be scaled in reverse order and it will be evaluated to `1e6` as shown below:

```
spotPrice = spotPrice * BalancerConstants.BALANCER_PRECISION / scaleFactor;
spotPrice = 1e18 * 1e18 / 1e30
```

```
spotPrice = 1e6
```

DAI/USDC Spot Price If it is the opposite where the primary token is USDC (6 decimals) and the secondary token is DAI (18 decimals), the calculation of the spot price will be as follows:

The `scaleFactor` at Line 47 will be evaluated to as follows:

```
scaleFactor = poolContext.secondaryScaleFactor *
↪  BalancerConstants.BALANCER_PRECISION / poolContext.primaryScaleFactor
scaleFactor = 1e18 * 1e18 / 1e30
scaleFactor = 1e6
```

Finally, the spot price will be scaled in reverse order and it will be evaluated to `1e30` as shown below:

```
spotPrice = spotPrice * BalancerConstants.BALANCER_PRECISION / scaleFactor;
spotPrice = 1e18 * 1e18 / 1e6
spotPrice = 1e30
```

Note about the spot price Assuming that the spot price of USDC and DAI is 1:1. As shown above, if the decimals of two tokens are not the same, the final spot price will end up either 1e6 (USDC/DAI) or 1e30 (DAI/USDC). However, if the decimals of two tokens (e.g. wstETH and WETH) are the same, this issue stays hidden as the `scaleFactor` in Line 47 will always be 1e18 as both `secondaryScaleFactor` and `primaryScaleFactor` cancel out each other.

It was observed that the spot price returned from the `Stable2TokenOracleMath._getSpotPrice` function is being compared with the oracle price from the `TwoTokenPoolUtils._getOraclePairPrice` function to determine if the pool has been manipulated within many functions.

```
uint256 oraclePrice =
↪  poolContext._getOraclePairPrice(strategyContext.tradingModule);
```

Based on the implementation of the `TwoTokenPoolUtils._getOraclePairPrice` function , the `oraclePrice` returned by this function is always denominated in 18 decimals regardless of the decimals of the underlying tokens. For instance, assume the spot price of USDC (6 decimals) and DAI (18 decimals) is 1:1. The spot price returned by this oracle function for USDC/DAI will be `1e18` and DAI/USDC will be `1e18`.

In many functions, the spot price returned from the `Stable2TokenOracleMath._getSpotPrice` function is compared with the oracle price via the `Stable2TokenOracleMath._checkPriceLimit`. Following is one such example. The `oraclePrice` will be `1e18`, while the `spotPrice` will be either `1e6` or `1e30` in our

example. This will cause the `_checkPriceLimit` to always revert because of the large discrepancy between the two prices.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L71

```
File: Stable2TokenOracleMath.sol
71:     function _getMinExitAmounts(
72:         StableOracleContext calldata oracleContext,
73:         TwoTokenPoolContext calldata poolContext,
74:         StrategyContext calldata strategyContext,
75:         uint256 oraclePrice,
76:         uint256 bptAmount
77:     ) internal view returns (uint256 minPrimary, uint256 minSecondary) {
78:         // Oracle price is always specified in terms of primary, so
 ↪  tokenIndex == 0 for primary
79:         // Validate the spot price to make sure the pool is not being
 ↪  manipulated
80:         uint256 spotPrice = _getSpotPrice({
81:             oracleContext: oracleContext,
82:             poolContext: poolContext,
83:             primaryBalance: poolContext.primaryBalance,
84:             secondaryBalance: poolContext.secondaryBalance,
85:             tokenIndex: 0
86:         });
87:         _checkPriceLimit(strategyContext, oraclePrice, spotPrice);
```

Other affected functions include the following:

- Stable2TokenOracleMath._validateSpotPriceAndPairPrice
- Stable2TokenOracleMath._getTimeWeightedPrimaryBalance

Impact

A vault supporting tokens with two different decimals will have many of its key functions will be broken as the `_checkPriceLimit` will always revert. For instance, rewards cannot be reinvested and vaults cannot be settled since they rely on the `_checkPriceLimit` function.

If the reward cannot be reinvested, the strategy tokens held by the users will not appreciate. If the vault cannot be settled, the vault debt cannot be repaid to Notional and the gain cannot be realized. Loss of assets for both users and Notional

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L15

Tool used

Manual Review

Recommendation

Within the `Stable2TokenOracleMath._getSpotPrice`, normalize the spot price back to 1e18 before returning the result. This ensures that it can be compared with the oracle price, which is denominated in 1e18 precision.

This has been implemented in the spot price function (`Boosted3TokenPoolUtils._getSpotPriceWithInvariant`) of another pool (`Boosted3Token`). However, it was not consistently applied in `TwoTokenPool`.

Discussion

weitianjie2000

valid, will fix

Issue H-2: Rounding differences when computing the invariant

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/17

Found by

xiaoming90

Summary

The invariant used within Boosted3Token vault to compute the spot price is not aligned with the Balancer's ComposableBoostedPool due to rounding differences. The spot price is used to verify if the pool has been manipulated before executing certain key vault actions (e.g. settle vault, reinvest rewards). In the worst-case scenario, it might potentially fail to detect the pool has been manipulated as the spot price computed might be inaccurate.

Vulnerability Detail

The Boosted3Token leverage vault relies on the old version of the `StableMath._calculateInvariant` that allows the caller to specify if the computation should round up or down via the `roundUp` parameter.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/StableMath.sol#L28

```
File: StableMath.sol
28:     function _calculateInvariant(
29:         uint256 amplificationParameter,
30:         uint256[] memory balances,
31:         bool roundUp
32:     ) internal pure returns (uint256) {
33:         /**********************************************************************
 ↪  **************************
34:         // invariant
 ↪                          //
35:         // D = invariant
 ↪  D^(n+1)                      //
36:         // A = amplification coefficient      A  n^n S + D = A D n^n +
 ↪  -----------                    //
37:         // S = sum of balances
 ↪  n^n P                       //
38:         // P = product of balances
 ↪                          //
```

```
39:         // n = number of tokens
↪                         //
40:         **********x****************************************************************
↪  **************************/
41:
42:         unchecked {
43:             // We support rounding up or down.
44:             uint256 sum = 0;
45:             uint256 numTokens = balances.length;
46:             for (uint256 i = 0; i < numTokens; i++) {
47:                 sum = sum.add(balances[i]);
48:             }
49:             if (sum == 0) {
50:                 return 0;
51:             }
52:
53:             uint256 prevInvariant = 0;
54:             uint256 invariant = sum;
55:             uint256 ampTimesTotal = amplificationParameter * numTokens;
56:
57:             for (uint256 i = 0; i < 255; i++) {
58:                 uint256 P_D = balances[0] * numTokens;
59:                 for (uint256 j = 1; j < numTokens; j++) {
60:                     P_D = Math.div(Math.mul(Math.mul(P_D, balances[j]),
↪  numTokens), invariant, roundUp);
61:                 }
62:                 prevInvariant = invariant;
63:                 invariant = Math.div(
64:                     Math.mul(Math.mul(numTokens, invariant), invariant).add(
65:                         Math.div(Math.mul(Math.mul(ampTimesTotal, sum),
↪  P_D), _AMP_PRECISION, roundUp)
66:                     ),
67:                     Math.mul(numTokens + 1, invariant).add(
68:                         // No need to use checked arithmetic for the amp
↪  precision, the amp is guaranteed to be at least 1
69:                         Math.div(Math.mul(ampTimesTotal - _AMP_PRECISION,
↪  P_D), _AMP_PRECISION, !roundUp)
70:                     ),
71:                     roundUp
72:                 );
73:
74:                 if (invariant > prevInvariant) {
75:                     if (invariant - prevInvariant <= 1) {
76:                         return invariant;
77:                     }
78:                 } else if (prevInvariant - invariant <= 1) {
79:                     return invariant;
80:                 }
```

```
81:             }
82:         }
83:
84:         revert CalculationDidNotConverge();
85:     }
```

Within the `Boosted3TokenPoolUtils._getSpotPrice` and `Boosted3TokenPoolUtils._getValidatedPoolData` functions, the `StableMath._calculateInvariant` is computed rounding up.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L15

```
File: Boosted3TokenPoolUtils.sol
76:     function _getSpotPrice(
77:         ThreeTokenPoolContext memory poolContext,
78:         BoostedOracleContext memory oracleContext,
79:         uint8 tokenIndex
80:     ) internal pure returns (uint256 spotPrice) {
..SNIP..
88:         uint256[] memory balances = _getScaledBalances(poolContext);
89:         uint256 invariant = StableMath._calculateInvariant(
90:             oracleContext.ampParam, balances, true // roundUp = true
91:         );
```

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L284

```
File: Boosted3TokenPoolUtils.sol
284:     function _getValidatedPoolData(
285:         ThreeTokenPoolContext memory poolContext,
286:         BoostedOracleContext memory oracleContext,
287:         StrategyContext memory strategyContext
288:     ) internal view returns (uint256 virtualSupply, uint256[] memory
↪  balances, uint256 invariant) {
289:         (virtualSupply, balances) =
290:             _getVirtualSupplyAndBalances(poolContext, oracleContext);
291:
292:         // Get the current and new invariants. Since we need a bigger new
↪  invariant, we round the current one up.
293:         invariant = StableMath._calculateInvariant(
294:             oracleContext.ampParam, balances, true // roundUp = true
295:         );
```

However, Balancer has since migrated its Boosted3Token pool from the legacy BoostedPool structure to a new ComposableBoostedPool contract.

The new ComposableBoostedPool contract uses a newer version of the StableMath library where the `StableMath._calculateInvariant` function always rounds down.

https://etherscan.io/address/0xa13a9247ea42d743238089903570127dda72fe44#code#F16#L57

```
function _calculateInvariant(uint256 amplificationParameter, uint256[] memory
↪  balances)
    internal
    pure
    returns (uint256)
{
    /**********************************************************************
↪  ******************
    // invariant
↪                   //
    // D = invariant                                                  D^(n+1)
↪                   //
    // A = amplification coefficient      A  n^n S + D = A D n^n + -----------
↪                   //
    // S = sum of balances                                             n^n P
↪                   //
    // P = product of balances
↪                   //
    // n = number of tokens
↪                   //
    **********************************************************************
↪  ******************/

    // Always round down, to match Vyper's arithmetic (which always truncates).

    uint256 sum = 0; // S in the Curve version
    uint256 numTokens = balances.length;
    for (uint256 i = 0; i < numTokens; i++) {
        sum = sum.add(balances[i]);
    }
    if (sum == 0) {
        return 0;
    }
    ..SNIP..
```

Thus, Notional round up when calculating the invariant while Balancer's ComposableBoostedPool round down when calculating the invariant. This inconsistency will result in a different invariant

Impact

The invariant is used to compute the spot price to verify if the pool has been manipulated before executing certain key vault actions (e.g. settle vault, reinvest rewards). If the inputted invariant is inaccurate, the spot price computed might not be accurate and might not match the actual spot price of the Balancer Pool. In the worst-case scenario, it might potentially fail to detect the pool has been manipulated and the trade proceeds to execute against the manipulated pool leading to a loss of assets.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/StableMath.sol#L28

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L15

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L284

Tool used

Manual Review

Recommendation

To avoid any discrepancy in the result, ensure that the StableMath library used by Balancer's ComposableBoostedPool and Notional's Boosted3Token leverage vault are aligned, and the implementation of the StableMath functions is the same between them.

Discussion

weitianjie2000

valid, will fix

Issue H-3: Users deposit assets to the vault but receives no strategy token in return

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/16

Found by

xiaoming90

Summary

Due to a rounding error in Solidity, it is possible that a user deposits assets to the vault, but receives no strategy token in return due to issues in the following functions:

- StrategyUtils._convertBPTClaimToStrategyTokens
- Boosted3TokenPoolUtils._deposit
- TwoTokenPoolUtils._deposit

Vulnerability Detail

This affects both the TwoToken and Boosted3Token vaults

```
int256 internal constant INTERNAL_TOKEN_PRECISION = 1e8;
uint256 internal constant BALANCER_PRECISION = 1e18;
```

Within the `StrategyUtils._convertBPTClaimToStrategyTokens` function, it was observed that the numerator precision (1e8) is much smaller than the denominator precision (1e18).

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/strategy/StrategyUtils.sol#L27

```
File: StrategyUtils.sol
26:     /// @notice Converts BPT to strategy tokens
27:     function _convertBPTClaimToStrategyTokens(StrategyContext memory
↪  context, uint256 bptClaim)
28:         internal pure returns (uint256 strategyTokenAmount) {
29:         if (context.vaultState.totalBPTHeld == 0) {
30:             // Strategy tokens are in 8 decimal precision, BPT is in 18.
↪  Scale the minted amount down.
31:             return (bptClaim * uint256(Constants.INTERNAL_TOKEN_PRECISION)) /
32:                 BalancerConstants.BALANCER_PRECISION;
33:         }
34:
```

```
35:         // BPT held in maturity is calculated before the new BPT tokens are
↪  minted, so this calculation
36:         // is the tokens minted that will give the account a corresponding
↪  share of the new bpt balance held.
37:         // The precision here will be the same as strategy token supply.
38:         strategyTokenAmount = (bptClaim *
↪  context.vaultState.totalStrategyTokenGlobal) /
↪  context.vaultState.totalBPTHeld;
39:     }
```

As a result, the `StrategyUtils._convertBPTClaimToStrategyTokens` function might return zero strategy tokens under the following two conditions:

If the `totalBPTHeld` is zero (First Deposit) If the `totalBPTHeld` is zero, the code at Line 31 will be executed, and the following formula is used:

```
strategyTokenAmount = (bptClaim * uint256(Constants.INTERNAL_TOKEN_PRECISION)) /
↪  BalancerConstants.BALANCER_PRECISION;
strategyTokenAmount = (bptClaim * 1e8) / 1e18
strategyTokenAmount = ((10 ** 10 - 1) * 1e8) / 1e18 = 0
```

During the first deposit, if the user deposits less than 1e10 BPT, Solidity will round down and `strategyTokenAmount` will be zero.

If the `totalBPTHeld` is larger than zero (Subsequently Deposits) If the `totalBPTHeld` is larger than zero, the code at Line 38 will be executed, and the following formula is used:

```
strategyTokenAmount = (bptClaim * context.vaultState.totalStrategyTokenGlobal) /
↪  context.vaultState.totalBPTHeld;
strategyTokenAmount = (bptClaim * (x * 1e8))/ (y * 1e18)
```

If the numerator is less than the denominator, the `strategyTokenAmount` will be zero.

Therefore, it is possible that the users deposited their minted BPT to the vault, but received zero strategy tokens in return.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L408

```
File: Boosted3TokenPoolUtils.sol
408:     function _deposit(
409:         ThreeTokenPoolContext memory poolContext,
410:         StrategyContext memory strategyContext,
411:         AuraStakingContext memory stakingContext,
412:         BoostedOracleContext memory oracleContext,
413:         uint256 deposit,
```

```
414:         uint256 minBPT
415:     ) internal returns (uint256 strategyTokensMinted) {
416:         uint256 bptMinted = poolContext._joinPoolAndStake({
417:             strategyContext: strategyContext,
418:             stakingContext: stakingContext,
419:             oracleContext: oracleContext,
420:             deposit: deposit,
421:             minBPT: minBPT
422:         });
423:
424:         strategyTokensMinted =
↪  strategyContext._convertBPTClaimToStrategyTokens(bptMinted);
425:
426:         strategyContext.vaultState.totalBPTHeld += bptMinted;
427:         // Update global supply count
428:         strategyContext.vaultState.totalStrategyTokenGlobal +=
↪  strategyTokensMinted.toUint80();
429:         strategyContext.vaultState.setStrategyVaultState();
430:     }
```

Proof-of-Concept Assume that Alice is the first depositor, and she forwarded 10000 BPT. During the first mint, the strategy token will be minted in a 1:1 ratio. Therefore, Alice will receive 10000 strategy tokens in return. At this point in time, `totalStrategyTokenGlobal` = 10000 strategy tokens and `totalBPTHeld` is 10000 BPT.

When Bob deposits to the vault after Alice, he will be subjected to the following formula:

```
strategyTokenAmount = (bptClaim * context.vaultState.totalStrategyTokenGlobal) /
↪  context.vaultState.totalBPTHeld;
strategyTokenAmount = (bptClaim * (10000 * 1e8))/ (10000 * 1e18)
strategyTokenAmount = (bptClaim * (1e12))/ (1e22)
```

If Bob deposits less than 1e10 BPT, Solidity will round down and `strategyTokenAmount` will be zero. Bob will receive no strategy token in return for his BPT.

Another side effect of this issue is that if Alice withdraws all her strategy tokens, she will get back all her 10000 BPT plus the BPT that Bob deposited earlier.

Impact

Loss of assets for the users as they deposited their assets but receive zero strategy tokens in return.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L408

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol#L153

Tool used

Manual Review

Recommendation

Consider reverting if zero strategy token is minted. This check has been implemented in many well-known vault designs as this is a commonly known issue (e.g. Solmate)

```
function _deposit(
    ThreeTokenPoolContext memory poolContext,
    StrategyContext memory strategyContext,
    AuraStakingContext memory stakingContext,
    BoostedOracleContext memory oracleContext,
    uint256 deposit,
    uint256 minBPT
) internal returns (uint256 strategyTokensMinted) {
    uint256 bptMinted = poolContext._joinPoolAndStake({
        strategyContext: strategyContext,
        stakingContext: stakingContext,
        oracleContext: oracleContext,
        deposit: deposit,
        minBPT: minBPT
    });

    strategyTokensMinted =
↪   strategyContext._convertBPTClaimToStrategyTokens(bptMinted);
+   require(strategyTokensMinted != 0, "zero strategy token minted");

    strategyContext.vaultState.totalBPTHeld += bptMinted;
    // Update global supply count
    strategyContext.vaultState.totalStrategyTokenGlobal +=
↪   strategyTokensMinted.toUint80();
    strategyContext.vaultState.setStrategyVaultState();
}
```

Discussion

weitianjie2000

valid, will fix

Issue H-4: Vault's `totalStrategyTokenGlobal` will not be in sync

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/15

Found by

xiaoming90

Summary

The `strategyContext.vaultState.totalStrategyTokenGlobal` variable that tracks the number of strategy tokens held in the vault will not be in sync and will cause accounting issues within the vault.

Vulnerability Detail

This affects both the TwoToken and Boosted3Token vaults

The `StrategyUtils._convertStrategyTokensToBPTClaim` function might return zero if a small number of `strategyTokenAmount` is passed into the function. If `(strategyTokenAmount * context.vaultState.totalBPTHeld)` is smaller than `context.vaultState.totalStrategyTokenGlobal`, the `bptClaim` will be zero.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/strategy/StrategyUtils.sol#L18

```
File: StrategyUtils.sol
17:     /// @notice Converts strategy tokens to BPT
18:     function _convertStrategyTokensToBPTClaim(StrategyContext memory
↪  context, uint256 strategyTokenAmount)
19:         internal pure returns (uint256 bptClaim) {
20:         require(strategyTokenAmount <=
↪  context.vaultState.totalStrategyTokenGlobal);
21:         if (context.vaultState.totalStrategyTokenGlobal > 0) {
22:             bptClaim = (strategyTokenAmount *
↪  context.vaultState.totalBPTHeld) /
↪  context.vaultState.totalStrategyTokenGlobal;
23:         }
24:     }
```

In Line 441 of the `Boosted3TokenPoolUtils._redeem` function below, if `bptClaim` is zero, it will return zero and exit the function immediately.

If a small number of `strategyTokens` is passed into the `_redeem` function and the `bptClaim` ends up as zero, the caller of the `_redeem` function will assume that all the `strategyTokens` have been redeemed.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L432

```
File: Boosted3TokenPoolUtils.sol
432:     function _redeem(
433:         ThreeTokenPoolContext memory poolContext,
434:         StrategyContext memory strategyContext,
435:         AuraStakingContext memory stakingContext,
436:         uint256 strategyTokens,
437:         uint256 minPrimary
438:     ) internal returns (uint256 finalPrimaryBalance) {
439:         uint256 bptClaim =
↪  strategyContext._convertStrategyTokensToBPTClaim(strategyTokens);
440:
441:         if (bptClaim == 0) return 0;
442:
443:         finalPrimaryBalance = _unstakeAndExitPool({
444:             stakingContext: stakingContext,
445:             poolContext: poolContext,
446:             bptClaim: bptClaim,
447:             minPrimary: minPrimary
448:         });
449:
450:         strategyContext.vaultState.totalBPTHeld -= bptClaim;
451:         strategyContext.vaultState.totalStrategyTokenGlobal -=
↪  strategyTokens.toUint80();
452:         strategyContext.vaultState.setStrategyVaultState();
453:     }
```

The following function shows an example of the caller of the `_redeem` function at Line 171 below accepting the zero value as it does not revert when the zero value is returned by the `_redeem` function. Thus, it will consider the small number of `strategyTokens` to be redeemed. Note that the `_redeemFromNotional` function calls the `_redeem` function under the hood.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/BaseStrategyVault.sol#L163

```
File: BaseStrategyVault.sol
163:     function redeemFromNotional(
164:         address account,
165:         address receiver,
166:         uint256 strategyTokens,
167:         uint256 maturity,
168:         uint256 underlyingToRepayDebt,
169:         bytes calldata data
170:     ) external onlyNotional returns (uint256 transferToReceiver) {
```

```
171:         uint256 borrowedCurrencyAmount = _redeemFromNotional(account,
↪  strategyTokens, maturity, data);
172:
173:         uint256 transferToNotional;
174:         if (account == address(this) || borrowedCurrencyAmount <=
↪  underlyingToRepayDebt) {
175:             // It may be the case that insufficient tokens were redeemed to
↪  repay the debt. If this
176:             // happens the Notional will attempt to recover the shortfall
↪  from the account directly.
177:             // This can happen if an account wants to reduce their leverage
↪  by paying off debt but
178:             // does not want to sell strategy tokens to do so.
179:             // The other situation would be that the vault is calling
↪  redemption to deleverage or
180:             // settle. In that case all tokens go back to Notional.
181:             transferToNotional = borrowedCurrencyAmount;
182:         } else {
183:             transferToNotional = underlyingToRepayDebt;
184:             unchecked { transferToReceiver = borrowedCurrencyAmount -
↪  underlyingToRepayDebt; }
185:         }
186:
187:         if (_UNDERLYING_IS_ETH) {
188:             if (transferToReceiver > 0)
↪  payable(receiver).transfer(transferToReceiver);
189:             if (transferToNotional > 0)
↪  payable(address(NOTIONAL)).transfer(transferToNotional);
190:         } else {
191:             if (transferToReceiver > 0)
↪  _UNDERLYING_TOKEN.checkTransfer(receiver, transferToReceiver);
192:             if (transferToNotional > 0)
↪  _UNDERLYING_TOKEN.checkTransfer(address(NOTIONAL), transferToNotional);
193:         }
194:     }
```

Subsequently, on Notional side, it will deduct the redeemed strategy tokens from its`vaultState.totalStrategyTokens` state (Refer to Line 177 below)

https://github.com/notional-finance/contracts-v2/blob/63eb0b46ec37e5fc5447bdde3d951dd90f245741/contracts/external/actions/VaultAction.sol#L157

```
File: VaultAction.sol
156:     /// @notice Redeems strategy tokens to cash
157:     function _redeemStrategyTokensToCashInternal(
158:         VaultConfig memory vaultConfig,
159:         uint256 maturity,
160:         uint256 strategyTokensToRedeem,
```

```
161:         bytes calldata vaultData
162:     ) private nonReentrant returns (int256 assetCashRequiredToSettle,
↪  int256 underlyingCashRequiredToSettle) {
163:         // If the vault allows further re-entrancy then set the status back
↪  to the default
164:         if (vaultConfig.getFlag(VaultConfiguration.ALLOW_REENTRANCY)) {
165:             reentrancyStatus = _NOT_ENTERED;
166:         }
167:
168:         VaultState memory vaultState =
↪  VaultStateLib.getVaultState(vaultConfig.vault, maturity);
169:         (int256 assetCashReceived, uint256 underlyingToReceiver) =
↪  vaultConfig.redeemWithoutDebtRepayment(
170:             vaultConfig.vault, strategyTokensToRedeem, maturity, vaultData
171:         );
172:         require(assetCashReceived > 0);
173:         // Safety check to ensure that the vault does not somehow receive
↪  tokens in this scenario
174:         require(underlyingToReceiver == 0);
175:
176:         vaultState.totalAssetCash =
↪  vaultState.totalAssetCash.add(uint256(assetCashReceived));
177:         vaultState.totalStrategyTokens =
↪  vaultState.totalStrategyTokens.sub(strategyTokensToRedeem);
178:         vaultState.setVaultState(vaultConfig.vault);
179:
180:         emit VaultRedeemStrategyToken(vaultConfig.vault, maturity,
↪  assetCashReceived, strategyTokensToRedeem);
181:         return _getCashRequiredToSettle(vaultConfig, vaultState, maturity);
182:     }
```

However, the main issue is that when a small number of `strategyTokens` are redeemed and `bptClaim` is zero, the `_redeem` function will exit at Line 441 immediately. Thus, the redeemed strategy tokens are not deducted from the `strategyContext.vaultState.totalStrategyTokenGlobal` accounting variable on the Vault side.

Thus, `strategyContext.vaultState.totalStrategyTokenGlobal` on the Vault side will not be in sync with the `vaultState.totalStrategyTokens` on the Notional side.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L432

```
File: Boosted3TokenPoolUtils.sol
432:     function _redeem(
433:         ThreeTokenPoolContext memory poolContext,
434:         StrategyContext memory strategyContext,
435:         AuraStakingContext memory stakingContext,
```

```
436:         uint256 strategyTokens,
437:         uint256 minPrimary
438:     ) internal returns (uint256 finalPrimaryBalance) {
439:         uint256 bptClaim =
↪  strategyContext._convertStrategyTokensToBPTClaim(strategyTokens);
440:
441:         if (bptClaim == 0) return 0;
442:
443:         finalPrimaryBalance = _unstakeAndExitPool({
444:             stakingContext: stakingContext,
445:             poolContext: poolContext,
446:             bptClaim: bptClaim,
447:             minPrimary: minPrimary
448:         });
449:
450:         strategyContext.vaultState.totalBPTHeld -= bptClaim;
451:         strategyContext.vaultState.totalStrategyTokenGlobal -=
↪  strategyTokens.toUint80();
452:         strategyContext.vaultState.setStrategyVaultState();
453:     }
```

Impact

The `strategyContext.vaultState.totalStrategyTokenGlobal` variable that tracks the number of strategy tokens held in the vault will not be in sync and will cause accounting issues within the vault. This means that the actual total strategy tokens in circulation and the `strategyContext.vaultState.totalStrategyTokenGlobal` will be different. The longer the issue is left unfixed, the larger the differences between them.

The `strategyContext.vaultState.totalStrategyTokenGlobal` will be larger than expected because it does not deduct the number of strategy tokens when it should be under certain conditions.

One example of the impact is as follows: The affected variable is used within the `_convertStrategyTokensToBPTClaim` and `_convertBPTClaimToStrategyTokens`, `_getBPTHeldInMaturity` functions. These functions are used within the deposit and redeem functions of the vault. Therefore, the number of strategy tokens or assets the users receive will not be accurate and might be less or more than expected.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/strategy/StrategyUtils.sol#L18

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L432

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol#L209

Tool used

Manual Review

Recommendation

The number of strategy tokens redeemed needs to be deducted from the vault's `totalStrategyTokenGlobal` regardless of the `bptClaim` value. Otherwise, the vault's `totalStrategyTokenGlobal` will not be in sync.

When `bptClaim` is zero, it does not always mean that no strategy token has been redeemed. Based on the current vault implementation, the `bptClaim` might be zero because the number of strategy tokens to be redeemed is too small and thus it causes Solidity to round down to zero.

```
function _redeem(
    ThreeTokenPoolContext memory poolContext,
    StrategyContext memory strategyContext,
    AuraStakingContext memory stakingContext,
    uint256 strategyTokens,
    uint256 minPrimary
) internal returns (uint256 finalPrimaryBalance) {
    uint256 bptClaim =
↪   strategyContext._convertStrategyTokensToBPTClaim(strategyTokens);
+   strategyContext.vaultState.totalStrategyTokenGlobal -=
↪   strategyTokens.toUint80();
+   strategyContext.vaultState.setStrategyVaultState();
+
    if (bptClaim == 0) return 0;

    finalPrimaryBalance = _unstakeAndExitPool({
        stakingContext: stakingContext,
        poolContext: poolContext,
        bptClaim: bptClaim,
        minPrimary: minPrimary
    });

    strategyContext.vaultState.totalBPTHeld -= bptClaim;
-   strategyContext.vaultState.totalStrategyTokenGlobal -=
↪   strategyTokens.toUint80();
    strategyContext.vaultState.setStrategyVaultState();
}
```

Discussion

weitianjie2000

valid, will fix

Issue H-5: Token amounts are scaled up twice causing the amounts to be inflated in two token vault

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/13

Found by

xiaoming90

Summary

Token amounts are scaled up twice causing the amounts to be inflated in two token vault when performing computation. This in turn causes the reinvest function to break leading to a loss of assets for vault users, and the value of their strategy tokens will be struck and will not appreciate.

Vulnerability Detail

In Line 121-124, the `primaryAmount` and `secondaryAmount` are scaled up to `BALANCER_PRECISION` (1e18). The reason for doing so is that balancer math functions expect all amounts to be in `BALANCER_PRECISION` (1e18).

Then, the scaled `primaryAmount` and `secondaryAmount` are passed into the `_getSpotPrice` function at Line 126.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L99

```
File: Stable2TokenOracleMath.sol
099:     function _validateSpotPriceAndPairPrice(
100:         StableOracleContext calldata oracleContext,
101:         TwoTokenPoolContext calldata poolContext,
102:         StrategyContext memory strategyContext,
103:         uint256 oraclePrice,
104:         uint256 primaryAmount,
105:         uint256 secondaryAmount
106:     ) internal view {
107:         // Oracle price is always specified in terms of primary, so
 ↪  tokenIndex == 0 for primary
108:         uint256 spotPrice = _getSpotPrice({
109:             oracleContext: oracleContext,
110:             poolContext: poolContext,
111:             primaryBalance: poolContext.primaryBalance,
112:             secondaryBalance: poolContext.secondaryBalance,
113:             tokenIndex: 0
114:         });
115:
```

```
116:         /// @notice Check spotPrice against oracle price to make sure that
117:         /// the pool is not being manipulated
118:         _checkPriceLimit(strategyContext, oraclePrice, spotPrice);
119:
120:         /// @notice Balancer math functions expect all amounts to be in
 ↪  BALANCER_PRECISION
121:         uint256 primaryPrecision = 10 ** poolContext.primaryDecimals;
122:         uint256 secondaryPrecision = 10 ** poolContext.secondaryDecimals;
123:         primaryAmount = primaryAmount *
 ↪  BalancerConstants.BALANCER_PRECISION / primaryPrecision;
124:         secondaryAmount = secondaryAmount *
 ↪  BalancerConstants.BALANCER_PRECISION / secondaryPrecision;
125:
126:         uint256 calculatedPairPrice = _getSpotPrice({
127:             oracleContext: oracleContext,
128:             poolContext: poolContext,
129:             primaryBalance: primaryAmount,
130:             secondaryBalance: secondaryAmount,
131:             tokenIndex: 0
132:         });
```

Within the `_getSpotPrice` function, the `primaryBalance` and `secondaryBalance` are scaled up again at Line 25 - 28. As such, any token (e.g. USDC) with a decimal of less than `BALANCER_PRECISION` (1e18) will be scaled up twice. This will cause the `balanceX` or `balanceY` to be inflated.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L15

```
File: Stable2TokenOracleMath.sol
15:     function _getSpotPrice(
16:         StableOracleContext memory oracleContext,
17:         TwoTokenPoolContext memory poolContext,
18:         uint256 primaryBalance,
19:         uint256 secondaryBalance,
20:         uint256 tokenIndex
21:     ) internal view returns (uint256 spotPrice) {
22:         require(tokenIndex < 2); /// @dev invalid token index
23:
24:         /// Apply scale factors
25:         uint256 scaledPrimaryBalance = primaryBalance *
 ↪  poolContext.primaryScaleFactor
26:             / BalancerConstants.BALANCER_PRECISION;
27:         uint256 scaledSecondaryBalance = secondaryBalance *
 ↪  poolContext.secondaryScaleFactor
28:             / BalancerConstants.BALANCER_PRECISION;
29:
```

```
30:         /// @notice poolContext balances are always in BALANCER_PRECISION
 ↪  (1e18)
31:         (uint256 balanceX, uint256 balanceY) = tokenIndex == 0 ?
32:             (scaledPrimaryBalance, scaledSecondaryBalance) :
33:             (scaledSecondaryBalance, scaledPrimaryBalance);
34:
35:         uint256 invariant = StableMath._calculateInvariant(
36:             oracleContext.ampParam, StableMath._balances(balanceX,
 ↪  balanceY), true // round up
37:         );
38:
39:         spotPrice = StableMath._calcSpotPrice({
40:             amplificationParameter: oracleContext.ampParam,
41:             invariant: invariant,
42:             balanceX: balanceX,
43:             balanceY: balanceY
44:         });
45:
46:         /// Apply secondary scale factor in reverse
47:         uint256 scaleFactor = tokenIndex == 0 ?
48:             poolContext.secondaryScaleFactor *
 ↪  BalancerConstants.BALANCER_PRECISION / poolContext.primaryScaleFactor :
49:             poolContext.primaryScaleFactor *
 ↪  BalancerConstants.BALANCER_PRECISION / poolContext.secondaryScaleFactor;
50:         spotPrice = spotPrice * BalancerConstants.BALANCER_PRECISION /
 ↪  scaleFactor;
51:     }
```

Balancer's Scaling Factors It is important to know the underlying mechanism of scaling factors within Balancer to understand this issue.

Within Balancer, all stable math calculations within the Balancer's pools are performed in 1e18. Thus, before passing the token balances to the stable math functions, all the balances need to be normalized to 18 decimals.

For instance, assume that 100 USDC needs to be passed into the stable math functions for some computation. 100 USDC is equal to `100e6` since the decimals of USDC is `6`. To normalize it to 18 decimals, 100 USDC (`100e6`) will be multiplied by its scaling factor (1e12), and the result will be `100e18`.

The following code taken from Balancer shows that the scaling factor is comprised of the scaling factor multiplied by the token rate. The scaling factor is the value needed to normalize the token balance to 18 decimals.

https://etherscan.io/address/0x32296969Ef14EB0c6d29669C550D4a0449130230#code

```
/**
```

```
 * @dev Overrides scaling factor getter to introduce the tokens' price rate.
 * Note that it may update the price rate cache if necessary.
 */
function _scalingFactors() internal view virtual override returns (uint256[]
↪  memory scalingFactors) {
    // There is no need to check the arrays length since both are based on
↪  `_getTotalTokens`
    // Given there is no generic direction for this rounding, it simply follows
↪  the same strategy as the BasePool.
    scalingFactors = super._scalingFactors();
    scalingFactors[0] = scalingFactors[0].mulDown(_priceRate(_token0));
    scalingFactors[1] = scalingFactors[1].mulDown(_priceRate(_token1));
}
```

Another point to note is that Balancer's stable math functions perform calculations in fixed point format. Therefore, the scaling factor will consist of the `FixedPoint.ONE` (1e18) multiplied by the value needed to normalize the token balance to 18 decimals. If it is a USDC with 6 decimals, the scaling factor will be `1e30`:

```
FixedPoint.ONE * 10**decimalsDifference
1e18 * 1e12 = 1e30
```

https://etherscan.io/address/0x32296969Ef14EB0c6d29669C550D4a0449130230#code

```
/**
 * @dev Returns a scaling factor that, when multiplied to a token amount for
↪  `token`, normalizes its balance as if
 * it had 18 decimals.
 */
function _computeScalingFactor(IERC20 token) internal view returns (uint256) {
    // Tokens that don't implement the `decimals` method are not supported.
    uint256 tokenDecimals = ERC20(address(token)).decimals();

    // Tokens with more than 18 decimals are not supported.
    uint256 decimalsDifference = Math.sub(18, tokenDecimals);
    return FixedPoint.ONE * 10**decimalsDifference;
}
```

Proof-of-Concept Assume that one of the tokens in Notional's two token leverage vault has a decimal of less than 18. Let's take USDC as an example.

1. 100 USDC (1e6) is passed into the `_validateSpotPriceAndPairPrice` function as the `primaryAmount`. In Line 121-124 of the `_validateSpotPriceAndPairPrice` function, the `primaryAmount` will be scaled up to `BALANCER_PRECISION` (1e18).

```
primaryAmount = primaryAmount * BalancerConstants.BALANCER_PRECISION /
↪  primaryPrecision;
primaryAmount = 100e6 * 1e18 / 1e6
primaryAmount = 100e18
```

2. Within the `_getSpotPrice` function, the `primaryBalance` is scaled up again at Line 25 - 28 of the `_getSpotPrice` function.

```
scaledPrimaryBalance = primaryBalance * poolContext.primaryScaleFactor /
↪  BalancerConstants.BALANCER_PRECISION;
scaledPrimaryBalance = 100e18 * 1e30 / 1e18
scaledPrimaryBalance = 1e30
scaledPrimaryBalance = 1000000000000e18
```

   As shown above, normalized 100 USDC (100e18) ended up becoming normalized 1000000000000 USDC (1000000000000e18). Therefore, the stable math functions are computed with an inflated balance of 1000000000000 USDC instead of 100 USDC.

Impact

The spot price computed by the `Stable2TokenOracleMath._getSpotPrice` function will deviate from the actual price because inflated balances were passed into it. The deviated spot price will then be passed to the `_checkPriceLimit` function to verify if the spot price has deviated from the oracle price. The check will fail and cause a revert. This will in turn cause the `Stable2TokenOracleMath._validateSpotPriceAndPairPrice` function to revert.

Therefore, any function that relies on the `Stable2TokenOracleMath._validateSpotPriceAndPairPrice` function will be affected. It was found that the `MetaStable2TokenAuraHelper.reinvestReward` relies on the `Stable2TokenOracleMath._validateSpotPriceAndPairPrice` function. As such, reinvest feature of the vault will be broken and the vault will not be able to reinvest its rewards.

This in turn led to a loss of assets for vault users, and the value of their strategy tokens will be struck and will not appreciate.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L99

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/math/Stable2TokenOracleMath.sol#L15

Tool used

Manual Review

Recommendation

Since the token balances are already normalized to 18 decimals within the `_getSpotPrice` function, the code to normalize the token balances in the `_validateSpotPriceAndPairPrice` function can be removed.

```
    function _validateSpotPriceAndPairPrice(
        StableOracleContext calldata oracleContext,
        TwoTokenPoolContext calldata poolContext,
        StrategyContext memory strategyContext,
        uint256 oraclePrice,
        uint256 primaryAmount,
        uint256 secondaryAmount
    ) internal view {
        // Oracle price is always specified in terms of primary, so tokenIndex
↪   == 0 for primary
        uint256 spotPrice = _getSpotPrice({
            oracleContext: oracleContext,
            poolContext: poolContext,
            primaryBalance: poolContext.primaryBalance,
            secondaryBalance: poolContext.secondaryBalance,
            tokenIndex: 0
        });

        /// @notice Check spotPrice against oracle price to make sure that
        /// the pool is not being manipulated
        _checkPriceLimit(strategyContext, oraclePrice, spotPrice);

-        /// @notice Balancer math functions expect all amounts to be in
↪   BALANCER_PRECISION
-        uint256 primaryPrecision = 10 ** poolContext.primaryDecimals;
-        uint256 secondaryPrecision = 10 ** poolContext.secondaryDecimals;
-        primaryAmount = primaryAmount * BalancerConstants.BALANCER_PRECISION /
↪   primaryPrecision;
-        secondaryAmount = secondaryAmount *
↪   BalancerConstants.BALANCER_PRECISION / secondaryPrecision;

        uint256 calculatedPairPrice = _getSpotPrice({
            oracleContext: oracleContext,
            poolContext: poolContext,
            primaryBalance: primaryAmount,
            secondaryBalance: secondaryAmount,
            tokenIndex: 0
        });
```

```
        /// @notice Check the calculated primary/secondary price against the
↪   oracle price
        /// to make sure that we are joining the pool proportionally
        _checkPriceLimit(strategyContext, oraclePrice, calculatedPairPrice);
    }
```

Discussion

weitianjie2000

valid, will fix

Issue H-6: `msgValue` will not be populated if ETH is the secondary token

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/12

Found by

xiaoming90

Summary

`msgValue` will not be populated if ETH is the secondary token in the two token leverage vault, leading to a loss of assets as the ETH is not forwarded to the Balancer Pool during a trade.

Vulnerability Detail

Based on the source code of the two token pool leverage vault, it is possible to deploy a vault to support a Balancer pool with an arbitrary token as the primary token and ETH as the secondary token. The primary token is always the borrowing currency in the vault.

However, Line 60 of `TwoTokenPoolUtils._getPoolParams` function below assumes that if one of the two tokens is ETH in the pool, it will always be the primary token or borrowing currency, which is not always the case. If the ETH is set as the secondary token, the `msgValue` will not be populated.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol#L45

```
File: TwoTokenPoolUtils.sol
44:     /// @notice Returns parameters for joining and exiting Balancer pools
45:     function _getPoolParams(
46:         TwoTokenPoolContext memory context,
47:         uint256 primaryAmount,
48:         uint256 secondaryAmount,
49:         bool isJoin
50:     ) internal pure returns (PoolParams memory) {
51:         IAsset[] memory assets = new IAsset[](2);
52:         assets[context.primaryIndex] = IAsset(context.primaryToken);
53:         assets[context.secondaryIndex] = IAsset(context.secondaryToken);
54:
55:         uint256[] memory amounts = new uint256[](2);
56:         amounts[context.primaryIndex] = primaryAmount;
57:         amounts[context.secondaryIndex] = secondaryAmount;
58:
```

```
59:         uint256 msgValue;
60:         if (isJoin && assets[context.primaryIndex] ==
↪  IAsset(Deployments.ETH_ADDRESS)) {
61:             msgValue = amounts[context.primaryIndex];
62:         }
63:
64:         return PoolParams(assets, amounts, msgValue);
65:     }
```

As a result, when the caller joins the Balancer pool, the `params.msgValue` will be empty, and no secondary token (ETH) will be forwarded to the Balancer pool. The ETH will remain stuck in the vault and the caller will receive much fewer BPT tokens in return.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/BalancerUtils.sol#L49

```
File: BalancerUtils.sol
48:     /// @notice Joins a balancer pool using exact tokens in
49:     function _joinPoolExactTokensIn(
50:         PoolContext memory context,
51:         PoolParams memory params,
52:         uint256 minBPT
53:     ) internal returns (uint256 bptAmount) {
54:         bptAmount = IERC20(address(context.pool)).balanceOf(address(this));
55:         Deployments.BALANCER_VAULT.joinPool{value: params.msgValue}(
56:             context.poolId,
57:             address(this),
58:             address(this),
59:             IBalancerVault.JoinPoolRequest(
60:                 params.assets,
61:                 params.amounts,
62:                 abi.encode(
63:                     IBalancerVault.JoinKind.EXACT_TOKENS_IN_FOR_BPT_OUT,
64:                     params.amounts,
65:                     minBPT // Apply minBPT to prevent front running
66:                 ),
67:                 false // Don't use internal balances
68:             )
69:         );
70:         bptAmount =
71:             IERC20(address(context.pool)).balanceOf(address(this)) -
72:             bptAmount;
73:     }
```

Impact

Loss of assets for the callers as ETH will remain stuck in the vault and not forwarded to the Balancer Pool. Since the secondary token (ETH) is not forwarded to the Balancer pool, the caller will receive much fewer BPT tokens in return when joining the pool.

This issue affects the deposit and reinvest reward functions of the vault, which means that the depositor will receive fewer strategy tokens in return during depositing, and the vault will receive less BPT in return during reinvesting.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol#L45

Tool used

Manual Review

Recommendation

Consider populating the `msgValue` if the secondary token is ETH.

```
/// @notice Returns parameters for joining and exiting Balancer pools
function _getPoolParams(
    TwoTokenPoolContext memory context,
    uint256 primaryAmount,
    uint256 secondaryAmount,
    bool isJoin
) internal pure returns (PoolParams memory) {
    IAsset[] memory assets = new IAsset[](2);
    assets[context.primaryIndex] = IAsset(context.primaryToken);
    assets[context.secondaryIndex] = IAsset(context.secondaryToken);

    uint256[] memory amounts = new uint256[](2);
    amounts[context.primaryIndex] = primaryAmount;
    amounts[context.secondaryIndex] = secondaryAmount;

    uint256 msgValue;
    if (isJoin && assets[context.primaryIndex] ==
↪   IAsset(Deployments.ETH_ADDRESS)) {
        msgValue = amounts[context.primaryIndex];
    }
+   if (isJoin && assets[context.secondaryIndex] ==
↪   IAsset(Deployments.ETH_ADDRESS)) {
+       msgValue = amounts[context.secondaryIndex];
```

```
+   }

    return PoolParams(assets, amounts, msgValue);
}
```

Discussion

weitianjie2000

valid, will fix

Issue H-7: `totalBPTSupply` will be excessively inflated

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/11

Found by

thekmj, xiaoming90

Summary

The `totalBPTSupply` will be excessively inflated as `totalSupply` was used instead of `virtualSupply`. This might cause a boosted balancer leverage vault not to be emergency settled in a timely manner and holds too large of a share of the liquidity within the pool, thus having problems exiting its position.

Vulnerability Detail

Balancer's Boosted Pool uses Phantom BPT where all pool tokens are minted at the time of pool creation and are held by the pool itself. Therefore, `virtualSupply` should be used instead of `totalSupply` to determine the amount of BPT supply in circulation.

However, within the `Boosted3TokenAuraVault.getEmergencySettlementBPTAmount` function, the `totalBPTSupply` at Line 169 is derived from the `totalSupply` instead of the `virtualSupply`. As a result, `totalBPTSupply` will be excessively inflated (`2**(111)`).

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/Boosted3TokenAuraVault.sol#L169

```
File: Boosted3TokenAuraVault.sol
165:     function getEmergencySettlementBPTAmount(uint256 maturity) external
↪  view returns (uint256 bptToSettle) {
166:         Boosted3TokenAuraStrategyContext memory context =
↪  _strategyContext();
167:         bptToSettle = context.baseStrategy._getEmergencySettlementParams({
168:             maturity: maturity,
169:             totalBPTSupply:
↪  IERC20(context.poolContext.basePool.basePool.pool).totalSupply()
170:         });
171:     }
```

As a result, the `emergencyBPTWithdrawThreshold` threshold will be extremely high. As such, the condition at Line 97 will always be evaluated as true and result in a revert.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/settlement/SettlementUtils.sol#L86

```
File: SettlementUtils.sol
86:     function _getEmergencySettlementParams(
87:         StrategyContext memory strategyContext,
88:         uint256 maturity,
89:         uint256 totalBPTSupply
90:     )  internal view returns(uint256 bptToSettle) {
91:         StrategyVaultSettings memory settings =
↪  strategyContext.vaultSettings;
92:         StrategyVaultState memory state = strategyContext.vaultState;
93:
94:         // Not in settlement window, check if BPT held is greater than
↪  maxBalancerPoolShare * total BPT supply
95:         uint256 emergencyBPTWithdrawThreshold =
↪  settings._bptThreshold(totalBPTSupply);
96:
97:         if (strategyContext.vaultState.totalBPTHeld <=
↪  emergencyBPTWithdrawThreshold)
98:             revert Errors.InvalidEmergencySettlement();
```

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/BalancerVaultStorage.sol#L52

```
File: BalancerVaultStorage.sol
52:     function _bptThreshold(StrategyVaultSettings memory
↪  strategyVaultSettings, uint256 totalBPTSupply)
53:         internal pure returns (uint256) {
54:         return (totalBPTSupply * strategyVaultSettings.maxBalancerPoolShare)
↪  / BalancerConstants.VAULT_PERCENT_BASIS;
55:     }
```

Impact

Anyone (e.g. off-chain keeper or bot) that relies on the `SettlementUtils.getEmergencySettlementBPTAmount` to determine if an emergency settlement is needed would be affected. The caller will presume that since the function reverts, emergency settlement is not required and the BPT threshold is still within the healthy level. The caller will wrongly decided not to perform an emergency settlement on a vault that has already exceeded the BPT threshold.

If a boosted balancer leverage vault is not emergency settled in a timely manner and holds too large of a share of the liquidity within the pool, it will have problems exiting its position.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/Boosted3TokenAuraVault.sol#L169

Tool used

Manual Review

Recommendation

Update the function to compute the `totalBPTSupply` from the virtual supply.

```
    function getEmergencySettlementBPTAmount(uint256 maturity) external view
↪   returns (uint256 bptToSettle) {
        Boosted3TokenAuraStrategyContext memory context = _strategyContext();
        bptToSettle = context.baseStrategy._getEmergencySettlementParams({
            maturity: maturity,
-           totalBPTSupply:
↪   IERC20(context.poolContext.basePool.basePool.pool).totalSupply()
+           totalBPTSupply:
↪   context.poolContext._getVirtualSupply(context.oracleContext)
        });
    }
```

Discussion

weitianjie2000

valid, will fix

Issue H-8: Users redeem strategy tokens but receives no assets in return

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/10

Found by

xiaoming90

Summary

Due to a rounding error in Solidity, it is possible that a user burns their strategy tokens, but receives no assets in return due to issues in the following functions:

- StrategyUtils._convertStrategyTokensToBPTClaim
- Boosted3TokenPoolUtils._redeem
- TwoTokenPoolUtils._redeem

Vulnerability Detail

This affects both the TwoToken and Boosted3Token vaults

```
int256 internal constant INTERNAL_TOKEN_PRECISION = 1e8;
uint256 internal constant BALANCER_PRECISION = 1e18;
```

Within the `StrategyUtils._convertStrategyTokensToBPTClaim` function, it was observed that if the numerator is smaller than the denominator, the `bptClaim` will be zero.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/strategy/StrategyUtils.sol#L18

```
File: StrategyUtils.sol
18:     function _convertStrategyTokensToBPTClaim(StrategyContext memory
↪  context, uint256 strategyTokenAmount)
19:         internal pure returns (uint256 bptClaim) {
20:         require(strategyTokenAmount <=
↪  context.vaultState.totalStrategyTokenGlobal);
21:         if (context.vaultState.totalStrategyTokenGlobal > 0) {
22:             bptClaim = (strategyTokenAmount *
↪  context.vaultState.totalBPTHeld) /
↪  context.vaultState.totalStrategyTokenGlobal;
23:         }
24:     }
```

When the `bptClaim` is zero, the function returns zero instead of reverting. Therefore, it is possible that a user redeems ("burns") their strategy tokens, but receives no assets in return because the number of strategy tokens redeemed by the user is too small.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L432

```
File: Boosted3TokenPoolUtils.sol
432:     function _redeem(
433:         ThreeTokenPoolContext memory poolContext,
434:         StrategyContext memory strategyContext,
435:         AuraStakingContext memory stakingContext,
436:         uint256 strategyTokens,
437:         uint256 minPrimary
438:     ) internal returns (uint256 finalPrimaryBalance) {
439:         uint256 bptClaim =
 ↪  strategyContext._convertStrategyTokensToBPTClaim(strategyTokens);
440:
441:         if (bptClaim == 0) return 0;
442:
443:         finalPrimaryBalance = _unstakeAndExitPool({
444:             stakingContext: stakingContext,
445:             poolContext: poolContext,
446:             bptClaim: bptClaim,
447:             minPrimary: minPrimary
448:         });
449:
450:         strategyContext.vaultState.totalBPTHeld -= bptClaim;
451:         strategyContext.vaultState.totalStrategyTokenGlobal -=
 ↪  strategyTokens.toUint80();
452:         strategyContext.vaultState.setStrategyVaultState();
453:     }
```

Impact

Loss of assets for the users as they burn their strategy tokens, but receive no assets in return.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L432

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol#L209

Tool used

Manual Review

Recommendation

Consider reverting if the assets (bptClaim) received is zero. This check has been implemented in many well-known vault designs as this is a commonly known issue (e.g. Solmate)

```
function _redeem(
    ThreeTokenPoolContext memory poolContext,
    StrategyContext memory strategyContext,
    AuraStakingContext memory stakingContext,
    uint256 strategyTokens,
    uint256 minPrimary
) internal returns (uint256 finalPrimaryBalance) {
    uint256 bptClaim =
↪   strategyContext._convertStrategyTokensToBPTClaim(strategyTokens);

-   if (bptClaim == 0) return 0;
+   require(bptClaim > 0, "zero asset")

    finalPrimaryBalance = _unstakeAndExitPool({
        stakingContext: stakingContext,
        poolContext: poolContext,
        bptClaim: bptClaim,
        minPrimary: minPrimary
    });

    strategyContext.vaultState.totalBPTHeld -= bptClaim;
    strategyContext.vaultState.totalStrategyTokenGlobal -=
↪   strategyTokens.toUint80();
    strategyContext.vaultState.setStrategyVaultState();
}
```

Discussion

weitianjie2000

valid, will fix

Issue H-9: Scaling factor of the wrapped token is incorrect

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/9

Found by

Jeiwan, xiaoming90

Summary

The scaling factor of the wrapped token within the Boosted3Token leverage vault is incorrect. Thus, all the computations within the leverage vault will be incorrect. This leads to an array of issues such as users being liquidated prematurely or users being able to borrow more than they are allowed to.

Vulnerability Detail

In Line 120, it calls the `getScalingFactors` function of the LinearPool to fetch the scaling factors of the LinearPool.

In Line 123, it computes the final scaling factor of the wrapped token by multiplying the main token's decimal scaling factor with the wrapped token rate, which is incorrect.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/mixins/Boosted3TokenPoolMixin.sol#L109

```
File: Boosted3TokenPoolMixin.sol
109:     function _underlyingPoolContext(ILinearPool underlyingPool) private
↪  view returns (UnderlyingPoolContext memory) {
110:         (uint256 lowerTarget, uint256 upperTarget) =
↪  underlyingPool.getTargets();
111:         uint256 mainIndex = underlyingPool.getMainIndex();
112:         uint256 wrappedIndex = underlyingPool.getWrappedIndex();
113:
114:         (
115:             /* address[] memory tokens */,
116:             uint256[] memory underlyingBalances,
117:             /* uint256 lastChangeBlock */
118:         ) =
↪  Deployments.BALANCER_VAULT.getPoolTokens(underlyingPool.getPoolId());
119:
120:         uint256[] memory underlyingScalingFactors =
↪  underlyingPool.getScalingFactors();
121:         // The wrapped token's scaling factor is not constant, but
↪  increases over time as the wrapped token increases in
```

```
122:         // value.
123:         uint256 wrappedScaleFactor = underlyingScalingFactors[mainIndex] *
↪  underlyingPool.getWrappedTokenRate() /
124:             BalancerConstants.BALANCER_PRECISION;
125:
126:         return UnderlyingPoolContext({
127:             mainScaleFactor: underlyingScalingFactors[mainIndex],
128:             mainBalance: underlyingBalances[mainIndex],
129:             wrappedScaleFactor: wrappedScaleFactor,
130:             wrappedBalance: underlyingBalances[wrappedIndex],
131:             virtualSupply: underlyingPool.getVirtualSupply(),
132:             fee: underlyingPool.getSwapFeePercentage(),
133:             lowerTarget: lowerTarget,
134:             upperTarget: upperTarget
135:         });
136:     }
```

Following is the source code of LinearPool taken from Balancer - https://github.com/balancer-labs/balancer-v2-monorepo/blob/master/pkg/pool-linear/contracts/LinearPool.sol

The correct way of calculating the final scaling factor of the wrapped token is to multiply the wrapped token's decimal scaling factor by the wrapped token rate as shown below:

```
scalingFactors[_wrappedIndex] =
↪  _scalingFactorWrappedToken.mulDown(_getWrappedTokenRate());
```

The `_scalingFactorWrappedToken` is the scaling factor that, when multiplied to a token amount, normalizes its balance as if it had 18 decimals. The `_getWrappedTokenRate` function returns the wrapped token rate.

It is important to note that the decimal scaling factor of the main and wrapped tokens are not always the same. Thus, they cannot be used interchangeably.

https://github.com/balancer-labs/balancer-v2-monorepo/blob/c3479e35f01e690fb71b2bf6b38a15cb92128586/pkg/pool-linear/contracts/LinearPool.sol#L504

https://github.com/balancer-labs/balancer-v2-monorepo/blob/c3479e35f01e690fb71b2bf6b38a15cb92128586/pkg/pool-linear/contracts/LinearPool.sol#L521

```
// Scaling factors

function _scalingFactor(IERC20 token) internal view virtual returns (uint256) {
    if (token == _mainToken) {
        return _scalingFactorMainToken;
    } else if (token == _wrappedToken) {
```

```
        // The wrapped token's scaling factor is not constant, but increases
↪   over time as the wrapped token
        // increases in value.
        return _scalingFactorWrappedToken.mulDown(_getWrappedTokenRate());
    } else if (token == this) {
        return FixedPoint.ONE;
    } else {
        _revert(Errors.INVALID_TOKEN);
    }
}

/**
 * @notice Return the scaling factors for all tokens, including the BPT.
 */
function getScalingFactors() public view virtual override returns (uint256[]
↪   memory) {
    uint256[] memory scalingFactors = new uint256[](_TOTAL_TOKENS);

    // The wrapped token's scaling factor is not constant, but increases over
↪   time as the wrapped token increases in
    // value.
    scalingFactors[_mainIndex] = _scalingFactorMainToken;
    scalingFactors[_wrappedIndex] =
↪   _scalingFactorWrappedToken.mulDown(_getWrappedTokenRate());
    scalingFactors[_BPT_INDEX] = FixedPoint.ONE;

    return scalingFactors;
}
```

Impact

Within the Boosted 3 leverage vault, the balances are scaled before passing them to the stable math function for computation since the stable math function only works with balances that have been normalized to 18 decimals. If the scaling factor is incorrect, all the computations within the leverage vault will be incorrect, which affects almost all the vault functions.

For instance, the `Boosted3TokenAuraVault.convertStrategyToUnderlying` function relies on the wrapped scaling factor for its computation under the hood. This function is utilized by Notional's `VaultConfiguration.calculateCollateralRatio` function to determine the value of the vault share when computing the collateral ratio. If the underlying result is wrong, the collateral ratio will be wrong too, and this leads to an array of issues such as users being liquidated prematurely or users being able to borrow more than they are allowed to.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/mixins/Boosted3TokenPoolMixin.sol#L109

Tool used

Manual Review

Recommendation

There is no need to manually calculate the final scaling factor of the wrapped token again within the code. This is because the wrapped token scaling factor returned by the `LinearPool.getScalingFactors()` function already includes the token rate. Refer to the Balancer's source code above for referen

```
function _underlyingPoolContext(ILinearPool underlyingPool) private view returns
↪  (UnderlyingPoolContext memory) {
    (uint256 lowerTarget, uint256 upperTarget) = underlyingPool.getTargets();
    uint256 mainIndex = underlyingPool.getMainIndex();
    uint256 wrappedIndex = underlyingPool.getWrappedIndex();

    (
        /* address[] memory tokens */,
        uint256[] memory underlyingBalances,
        /* uint256 lastChangeBlock */
    ) = Deployments.BALANCER_VAULT.getPoolTokens(underlyingPool.getPoolId());

    uint256[] memory underlyingScalingFactors =
↪  underlyingPool.getScalingFactors();
-   // The wrapped token's scaling factor is not constant, but increases over
↪  time as the wrapped token increases in
-   // value.
-   uint256 wrappedScaleFactor = underlyingScalingFactors[mainIndex] *
↪  underlyingPool.getWrappedTokenRate() /
-       BalancerConstants.BALANCER_PRECISION;

    return UnderlyingPoolContext({
        mainScaleFactor: underlyingScalingFactors[mainIndex],
        mainBalance: underlyingBalances[mainIndex],
-       wrappedScaleFactor: wrappedScaleFactor,
+       wrappedScaleFactor: underlyingScalingFactors[wrappedIndex],
        wrappedBalance: underlyingBalances[wrappedIndex],
        virtualSupply: underlyingPool.getVirtualSupply(),
        fee: underlyingPool.getSwapFeePercentage(),
        lowerTarget: lowerTarget,
        upperTarget: upperTarget
```

```
    });
}
```

Discussion

weitianjie2000

valid, will fix

Issue M-1: Boosted3TokenPoolUtils.sol : _redeem - updating the `totalBPTHeld , totalStrategyTokenGlobal` after `_unstakeAndExitPool` is not safe

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/30

Found by

ak1

Summary

_redeem function is used to claim the BPT amount using the strategy tokens.

It is first calling the `_unstakeAndExitPool` function and then updating the `totalBPTHeld , totalStrategyTokenGlobal`

Vulnerability Detail

```
    function _redeem(
    ThreeTokenPoolContext memory poolContext,
    StrategyContext memory strategyContext,
    AuraStakingContext memory stakingContext,
    uint256 strategyTokens,
    uint256 minPrimary
) internal returns (uint256 finalPrimaryBalance) {
    uint256 bptClaim =
    ↪  strategyContext._convertStrategyTokensToBPTClaim(strategyTokens);


    if (bptClaim == 0) return 0;


    finalPrimaryBalance = _unstakeAndExitPool({
        stakingContext: stakingContext,
        poolContext: poolContext,
        bptClaim: bptClaim,
        minPrimary: minPrimary
    });


    strategyContext.vaultState.totalBPTHeld -= bptClaim;
    strategyContext.vaultState.totalStrategyTokenGlobal -=
    ↪  strategyTokens.toUint80();
```

```
    strategyContext.vaultState.setStrategyVaultState();
}
```

First _unstakeAndExitPool is called and then totalBPTHeld and totalStrategyTokenGlobal are updated.

Impact

Reentering during any of the function call inside `_unstakeAndExitPool` could be problematic. `stakingContext.auraRewardPool.withdrawAndUnwrap(bptClaim, false)` `BalancerUtils._swapGivenIn`

Well it need deep study to analyze the impact, but I would suggest to update the balance first and then call the `_unstakeAndExitPool`

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/internal/pool/Boosted3TokenPoolUtils.sol#L432-L453

Tool used

Manual Review

Recommendation

First update `totalBPTHeld and totalStrategyTokenGlobal` and then call the `_unstakeAndExitPool`

Discussion

weitianjie2000

invalid: we have disallow re-entrancy on the vault controller side

weitianjie2000

valid: we've decided to make the fix just in case

jeffywu

Agree, we can mark this issue as valid although I disagree with the High severity rating here. There is no clear re-entrancy vector that has been articulated and the code being called in _unstakeAndExitPool is both known and trusted (both Balancer and Aura).

hrishibhat

Agree with the sponsor about the issue being medium as there is no re-entrancy attack described in the issue. Considering this a medium severity issue.

Issue M-2: Unable to deploy new leverage vault for certain MetaStable Pool

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/20

Found by

Jeiwan, xiaoming90

Summary

Notional might have an issue deploying the new leverage vault for a MetaStable Pool that does not have Balancer Oracle enabled.

Vulnerability Detail

During deployment, the `MetaStable2TokenVaultMixin` contract checks if the MetaStable Pool's oracle is enabled. However, the Balancer Oracle has been deprecated (https://docs.balancer.fi/products/oracles-deprecated). In addition, Notional is no longer relying on Balancer Oracle in this iteration. Therefore, there is no need to check if the Balancer Oracle is enabled on the MetaStable Pool.

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/mixins/MetaStable2TokenVaultMixin.sol#L11

```
File: MetaStable2TokenVaultMixin.sol
11: abstract contract MetaStable2TokenVaultMixin is TwoTokenPoolMixin {
12:     constructor(NotionalProxy notional_, AuraVaultDeploymentParams memory
↪  params)
13:         TwoTokenPoolMixin(notional_, params)
14:     {
15:         // The oracle is required for the vault to behave properly
16:         (/* */, /* */, /* */, /* */, bool oracleEnabled) =
17:
↪  IMetaStablePool(address(BALANCER_POOL_TOKEN)).getOracleMiscData();
18:         require(oracleEnabled);
19:     }
```

Impact

Notional might have an issue deploying the new leverage vault for a MetaStable Pool that does not have Balancer Oracle enabled. Since Balancer Oracle has been deprecated, the Balancer Oracle will likely be disabled on the MetaStable Pool.

Code Snippet

https://github.com/sherlock-audit/2022-12-notional/blob/main/contracts/vaults/balancer/mixins/MetaStable2TokenVaultMixin.sol#L11

Tool used

Manual Review

Recommendation

Remove the Balancer Oracle check from the constructor.

```
    constructor(NotionalProxy notional_, AuraVaultDeploymentParams memory params)
        TwoTokenPoolMixin(notional_, params)
    {
-       // The oracle is required for the vault to behave properly
-       (/* */, /* */, /* */, /* */, bool oracleEnabled) =
-           IMetaStablePool(address(BALANCER_POOL_TOKEN)).getOracleMiscData();
-       require(oracleEnabled);
    }
```

Discussion

weitianjie2000

valid, will fix

Issue M-3: Possible division by zero depending on `TradingModule.g` return values

Source: https://github.com/sherlock-audit/2022-12-notional-judging/issues/2

Found by

ak1, MalfurionWhitehat, Deivitto

Summary

Some functions depending on `TradingModule.getOraclePrice` accept non-negative `(int256 answer, int256 decimals)` return values. In case any of those are equal to zero, division depending on `answer` or `decimals` will revert. In the worst case scenario, this will prevent the protocol from continuing operating.

Vulnerability Detail

The function `TradingModule.getOraclePrice` properly validates that return values from Chainlink price feeds are positive.

Nevertheless, `answer` may *currently* return zero, as it is calculated as `(basePrice * quoteDecimals * RATE_DECIMALS) / (quotePrice * baseDecimals);`, which can be truncated down to zero, depending on base/quote prices [1]. Additionally, `decimals` may *in the future* return zero, depending on changes to the protocol code, as the NatSpec states that this is a `number of decimals in the rate, currently hardcoded to 1e18` [2].

If any of these return values are zero, calculations that use division depending on `TradingModule.getOraclePrice` will revert.

More specifically:

[1]

1.1 TradingModule.getLimitAmount

```
require(oraclePrice >= 0); /// @dev Chainlink rate error
```

that calls `TradingUtils._getLimitAmount`, which reverts if oraclePrice

```
oraclePrice = (oracleDecimals * oracleDecimals) / oraclePrice;
```

[2] 2.1 TwoTokenPoolUtils._getOraclePairPrice

```
require(decimals >= 0);
```

```
if (uint256(decimals) != BalancerConstants.BALANCER_PRECISION) {
    rate = (rate * int256(BalancerConstants.BALANCER_PRECISION)) / decimals;
}
```

2.2 TradingModule.getLimitAmount

```
require(oracleDecimals >= 0); /// @dev Chainlink decimals error
```

that calls `TradingUtils._getLimitAmount`, which reverts if oracleDecimals

```
limitAmount =
    ((oraclePrice +
        ((oraclePrice * uint256(slippageLimit)) /
            Constants.SLIPPAGE_LIMIT_PRECISION)) * amount) /
    oracleDecimals;
```

2.3 CrossCurrencyfCashVault.convertStrategyToUnderlying

```
return (pvInternal * borrowTokenDecimals * rate) /
    (rateDecimals * int256(Constants.INTERNAL_TOKEN_PRECISION));
```

Impact

In the worst case, the protocol might stop operating.

Albeit unlikely that `decimals` is ever zero, since currently this is a hardcoded value, it is possible that `answer` might be zero due to round-down performed by the division in `TradingModule.getOraclePrice`. This can happen if the quote token is much more expensive than the base token. In this case, `TradingModule.getLimitAmount` and depending calls, such as `TradingModule.executeTradeWithDynamicSlippage` might revert.

Code Snippet

```
answer =
    (basePrice * quoteDecimals * RATE_DECIMALS) /
    (quotePrice * baseDecimals);
decimals = RATE_DECIMALS;
```

Tool used

Manual Review

Recommendation

Validate that the return values are strictly positive (instead of non-negative) in case depending function calculations may result in division by zero. This can be either done on `TradingModule.getOraclePrice` directly or on the depending functions.

```
diff --git a/contracts/trading/TradingModule.sol
 ↪  b/contracts/trading/TradingModule.sol
index bfc8505..70b40f2 100644
--- a/contracts/trading/TradingModule.sol
+++ b/contracts/trading/TradingModule.sol
@@ -251,6 +251,9 @@ contract TradingModule is Initializable, UUPSUpgradeable,
 ↪  ITradingModule {
             (basePrice * quoteDecimals * RATE_DECIMALS) /
             (quotePrice * baseDecimals);
         decimals = RATE_DECIMALS;
+
+        require(answer > 0); /// @dev Chainlink rate error
+        require(decimals > 0); /// @dev Chainlink decimals error
     }

     function _hasPermission(uint32 flags, uint32 flagID) private pure returns
 ↪  (bool) {
@@ -279,9 +282,6 @@ contract TradingModule is Initializable, UUPSUpgradeable,
 ↪  ITradingModule {
         // prettier-ignore
         (int256 oraclePrice, int256 oracleDecimals) = getOraclePrice(sellToken,
 ↪  buyToken);

-        require(oraclePrice >= 0); /// @dev Chainlink rate error
-        require(oracleDecimals >= 0); /// @dev Chainlink decimals error
-
         limitAmount = TradingUtils._getLimitAmount({
             tradeType: tradeType,
             sellToken: sellToken,
diff --git a/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol
 ↪  b/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol
index 4954c59..6315c0a 100644
--- a/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol
+++ b/contracts/vaults/balancer/internal/pool/TwoTokenPoolUtils.sol
@@ -76,10 +76,7 @@ library TwoTokenPoolUtils {
         (int256 rate, int256 decimals) = tradingModule.getOraclePrice(
             poolContext.primaryToken, poolContext.secondaryToken
         );
-        require(rate > 0);
-        require(decimals >= 0);

         if (uint256(decimals) != BalancerConstants.BALANCER_PRECISION) {
```

```
            rate = (rate * int256(BalancerConstants.BALANCER_PRECISION)) /
↪   decimals;
        }
```

Discussion

weitianjie2000

valid, will fix