



**SHERLOCK**

# **SHERLOCK SECURITY REVIEW FOR**



<b>Prepared for:</b>	<b>Notional</b>
<b>Prepared by:</b>	<b>Sherlock</b>
<b>Lead Security Expert:</b>	<b><u>xiaoming90</u></b>
<b>Dates Audited:</b>	<b>November 18 - November 25, 2023</b>
<b>Prepared on:</b>	<b>December 18, 2023</b>

## Introduction

Maximum Returns. Minimum Risk. Lend, borrow, and earn leveraged yield with DeFi's leading fixed rate lending protocol.

## Scope

Repository: notional-finance/leveraged-vaults

Branch: tests/composable-pool

Commit: cf17af56ca489bc2729e8a09567af36a0c723192

---

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
9	9

## Issues not fixed or acknowledged

Medium	High
0	0

## Security experts who found valid issues

[xiaoming90](#)  
[mstpr-brainbot](#)  
[Vagner](#)  
[lemonmon](#)

[coffiasd](#)  
[Tri-pathi](#)  
[shealtielanz](#)  
[AuditorPraise](#)

[bin2chen](#)  
[tvdung94](#)  
[0xMaroutis](#)  
[ZanyBonzy](#)



SHERLOCK

## Issue H-1: Rounding differences when computing the invariant

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/77>

### Found by

Tri-pathi, lemonmon, shealtielanz, xiaoming90

### Summary

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. 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.

### Vulnerability Detail

The Balancer's Composable Pool codebase 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/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/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 BalancerSpotPrice.\_calculateStableMathSpotPrice function, the StableMath.\_calculateInvariant is computed rounding up per Line 90 below

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L90>

```

File: BalancerSpotPrice.sol
78:     function _calculateStableMathSpotPrice(
79:         uint256 ampParam,
80:         uint256[] memory scalingFactors,
81:         uint256[] memory balances,
82:         uint256 scaledPrimary,
83:         uint256 primaryIndex,
84:         uint256 index2
85:     ) internal pure returns (uint256 spotPrice) {
86:         // Apply scale factors
87:         uint256 secondary = balances[index2] * scalingFactors[index2] /
↳ BALANCER_PRECISION;
88:
89:         uint256 invariant = StableMath._calculateInvariant(
90:             ampParam, StableMath._balances(scaledPrimary, secondary), true
↳ // round up
91:         );

```

The new Composable Pool contract uses a newer version of the StableMath library where the StableMath.\_calculateInvariant function always rounds down. Following is the StableMath.sol of one of the popular composable pools in Arbitrum that uses the new StableMath library

<https://arbiscan.io/address/0xade4a71bb62bec25154cfc7e6ff49a513b491e81#code#F28#L57> (Balancer rETH-WETH Stable Pool - Arbitrum)

```

function _calculateInvariant(uint256 amplificationParameter, uint256[] memory
↳ balances)
    internal
    pure
    returns (uint256)
{
    /*****
↳ *****/

```



```

    // invariant
    ↪          //
    // D = invariant                                 $D^{(n+1)}$ 
    ↪          //
    // A = amplification coefficient                 $A \cdot n^n S + D = A \cdot D \cdot n^n + \text{-----}$ 
    ↪          //
    // 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).
    ..SNIP..

```

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

## Impact

High, as per past contest's risk rating -

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

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. 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/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/math/StableMath.sol#L28>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L90>

## Tool used

Manual Review



## Recommendation

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

## Discussion

**jeffyu**

Valid issue

**jeffyu**

To some extent this is a duplicate of #83 which is a more complete description of the problem, but will leave that to the judges. It appears that Balancer has tweaked their `_calculateInvariant` implementation in the ComposableStablePool rewrite which included this rounding difference as well as other differences.

**jeffyu**

<https://github.com/notional-finance/leveraged-vaults/pull/63>

**xiaoming9090**

Fixed in [PR 63](#)



## Issue H-2: Incorrect scaling of the spot price

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/79>

### Found by

mstpr-brainbot, xiaoming90

### Summary

The incorrect scaling of the spot price leads to the incorrect spot price, which is later compared with the oracle price.

If the spot price is incorrect, it might potentially fail to detect the pool has been manipulated or result in unintended reverts due to false positives. In the worst-case scenario, the trade proceeds to execute against the manipulated pool, leading to a loss of assets.

### Vulnerability Detail

Per the comment and source code at Lines 97 to 103, the `SPOT_PRICE.getComposableSpotPrices` is expected to return the spot price in native decimals.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L97>

```
File: BalancerComposableAuraVault.sol
090:     function _checkPriceAndCalculateValue() internal view override returns
    ↪ (uint256) {
091:         (uint256[] memory balances, uint256[] memory spotPrices) =
    ↪ SPOT_PRICE.getComposableSpotPrices(
092:             BALANCER_POOL_ID,
093:             address(BALANCER_POOL_TOKEN),
094:             PRIMARY_INDEX()
095:         );
096:
097:         // Spot prices are returned in native decimals, convert them all to
    ↪ POOL_PRECISION
098:         // as required in the _calculateLPTokenValue method.
099:         (/* */, uint8[] memory decimals) = TOKENS();
100:         for (uint256 i; i < spotPrices.length; i++) {
101:             spotPrices[i] = spotPrices[i] * POOL_PRECISION() / 10 **
    ↪ decimals[i];
102:         }
103:
```





```
104:         return _calculateLPTokenValue(balances, spotPrices);
105:     }
```

Within the `getComposableSpotPrices` function, it will trigger the `_calculateStableMathSpotPrice` function. When the primary and secondary balances are passed into the `StableMath._calculateInvariant` and `StableMath._calcSpotPrice` functions, they are scaled up to 18 decimals precision as `StableMath` functions only work with balances that have been normalized to 18 decimals.

Assuming that the following states:

- Primary Token = USDC (6 decimals)
- Secondary Token = DAI (18 decimals)
- Primary Balance = 100 USDC (=100 \* 1e6)
- Secondary Balance = 100 DAI (=100 \* 1e18)
- `scalingFactors[USDC] = 1e12 * Fixed.ONE (1e18) = 1e30`
- `scalingFactors[DAI] = 1e0 * Fixed.ONE (1e18) = 1e18`
- The price between USDC and DAI is 1:1

After scaling the primary and secondary balances, the scaled balances will be as follows:

```
scaledPrimary = balances[USDC] * scalingFactors[USDC] / BALANCER_PRECISION
scaledPrimary = 100 * 1e6 * 1e30 / 1e18
scaledPrimary = 100 * 1e18

scaledSecondary = balances[DAI] * scalingFactors[DAI] / BALANCER_PRECISION
scaledSecondary = 100 * 1e18 * 1e18 / 1e18
scaledSecondary = 100 * 1e18
```

The spot price returned from the `StableMath._calcSpotPrice` function at Line 93 will be 1e18 (1:1).

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L93>

```
File: BalancerSpotPrice.sol
78:     function _calculateStableMathSpotPrice(
79:         uint256 ampParam,
80:         uint256[] memory scalingFactors,
81:         uint256[] memory balances,
82:         uint256 scaledPrimary,
83:         uint256 primaryIndex,
```



```

84:         uint256 index2
85:     ) internal pure returns (uint256 spotPrice) {
86:         // Apply scale factors
87:         uint256 secondary = balances[index2] * scalingFactors[index2] /
↳ BALANCER_PRECISION;
88:
89:         uint256 invariant = StableMath._calculateInvariant(
90:             ampParam, StableMath._balances(scaledPrimary, secondary), true
↳ // round up
91:         );
92:
93:         spotPrice = StableMath._calcSpotPrice(ampParam, invariant,
↳ scaledPrimary, secondary);
94:
95:         // Remove scaling factors from spot price
96:         spotPrice = spotPrice * scalingFactors[primaryIndex] /
↳ scalingFactors[index2];
97:     }

```

Subsequently, in Line 96 above, the code attempts to remove the scaling factor from the spot price (1e18).

```

spotPrice = spotPrice * scalingFactors[USDC] / scalingFactors[DAI];
spotPrice = 1e18 * 1e30 / 1e18
spotPrice = 1e30
spotPrice = 1e12 * 1e18

```

The spotPrice[DAI-Secondary] is not denominated in native precision after the scaling. The SPOT\_PRICE.getComposableSpotPrices will return the following spot prices:

```

spotPrice[USDC-Primary] = 0
spotPrice[DAI-Secondary] = 1e12 * 1e18

```

The returned spot prices will be scaled to POOL\_PRECISION (1e18). After the scaling, the spot price remains the same:

```

spotPrice[DAI-Secondary] = spotPrice[DAI-Secondary] * POOL_PRECISION /
↳ DAI_Decimal
spotPrice[DAI-Secondary] = 1e12 * 1e18 * 1e18 / 1e18
spotPrice[DAI-Secondary] = 1e12 * 1e18

```

The converted spot prices will be passed into the \_calculateLPTokenValue function. Within the \_calculateLPTokenValue function, the oracle price for DAI<>USDC will be 1e18. From here, the spotPrice[DAI-Secondary] (1e12 \* 1e18) is significantly



different from the oracle price (1e18), which will cause the pool manipulation check to revert.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L97>

```
File: BalancerComposableAuraVault.sol
090:     function _checkPriceAndCalculateValue() internal view override returns
    ↪ (uint256) {
091:         (uint256[] memory balances, uint256[] memory spotPrices) =
    ↪ SPOT_PRICE.getComposableSpotPrices(
092:             BALANCER_POOL_ID,
093:             address(BALANCER_POOL_TOKEN),
094:             PRIMARY_INDEX()
095:         );
096:
097:         // Spot prices are returned in native decimals, convert them all to
    ↪ POOL_PRECISION
098:         // as required in the _calculateLPTokenValue method.
099:         (/* */, uint8[] memory decimals) = TOKENS();
100:         for (uint256 i; i < spotPrices.length; i++) {
101:             spotPrices[i] = spotPrices[i] * POOL_PRECISION() / 10 **
    ↪ decimals[i];
102:         }
103:
104:         return _calculateLPTokenValue(balances, spotPrices);
105:     }
```

## Impact

The spot price is used to verify if the pool has been manipulated before executing certain key vault actions (e.g. reinvest rewards).

If the spot price is incorrect, it might potentially result in the following:

- Failure to detect the pool has been manipulated, resulting in the trade to execute against the manipulated pool, leading to a loss of assets.
- Unintended reverts due to false positives, breaking core functionalities of the protocol that rely on the `_checkPriceAndCalculateValue` function.

The affected `_checkPriceAndCalculateValue` function was found to be used within the following functions:

- `reinvestReward` - If the `_checkPriceAndCalculateValue` function is malfunctioning or reverts unexpectedly, the protocol will not be able to reinvest, leading to a loss of value for the vault shareholders.



- `convertStrategyToUnderlying` - This function is used by Notional V3 for the purpose of computing the collateral values and the account's health factor. If the `_checkPriceAndCalculateValue` function reverts unexpectedly due to an incorrect invariant/spot price, many of Notional's core functions will break. In addition, the collateral values and the account's health factor might be inflated if it fails to detect a manipulated pool due to incorrect invariant/spot price, potentially allowing the malicious actors to drain the main protocol.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L97>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L93>

## Tool used

Manual Review

## Recommendation

The spot price returned from `StableMath._calcSpotPrice` is denominated in `1e18` (`POOL_PRECISION`) since the inputted balances are normalized to 18 decimals. The scaling factors are used to normalize a balance to 18 decimals. By dividing or scaling down the spot price by the scaling factor, the native spot price will be returned.

```
spotPrice[DAI-Secondary] = spotPrice[DAI-Secondary] * Fixed.ONE /  
↳ scalingFactors[DAI];  
spotPrice = 1e18 * Fixed.ONE / (1e0 * Fixed.ONE)  
spotPrice = 1e18 * 1e18 / (1e0 * 1e18)  
spotPrice = 1e18
```

## Discussion

**jeffyu**

Valid issue

**jeffyu**

<https://github.com/notional-finance/leveraged-vaults/pull/63>

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/63>



**xiaoming9090**

Fixed in PR 63



## Issue H-3: Incorrect Spot Price

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/81>

### Found by

xiaoming90

### Summary

Multiple discrepancies between the implementation of Leverage Vault's `_calcSpotPrice` function and SDK were observed, which indicate that the computed spot price is incorrect.

If the spot price is incorrect, it might potentially fail to detect the pool has been manipulated. In the worst-case scenario, the trade proceeds to execute against the manipulated pool, leading to a loss of assets.

### Vulnerability Detail

The `BalancerSpotPrice._calculateStableMathSpotPrice` function relies on the `StableMath._calcSpotPrice` to compute the spot price of two tokens.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L93>

```
File: BalancerSpotPrice.sol
78:     function _calculateStableMathSpotPrice(
    ..SNIP..
86:         // Apply scale factors
87:         uint256 secondary = balances[index2] * scalingFactors[index2] /
    ↪ BALANCER_PRECISION;
88:
89:         uint256 invariant = StableMath._calculateInvariant(
90:             ampParam, StableMath._balances(scaledPrimary, secondary), true
    ↪ // round up
91:         );
92:
93:         spotPrice = StableMath._calcSpotPrice(ampParam, invariant,
    ↪ scaledPrimary, secondary);
```

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/math/StableMath.sol#L90>

```
File: StableMath.sol
087:     /**
```



```

088:     * @dev Calculates the spot price of token Y in token X.
089:     */
090:     function _calcSpotPrice(
091:         uint256 amplificationParameter,
092:         uint256 invariant,
093:         uint256 balanceX,
094:         uint256 balanceY
095:     ) internal pure returns (uint256) {
096:         /*****
↳      *****/
097:         //
↳      //
098:         //      2.a.x.y + a.y^2 + b.y
↳      //
099:         // spot price Y/X = - dx/dy = -----
↳      //
100:         //      2.a.x.y + a.x^2 + b.x
↳      //
101:         //
↳      //
102:         // n = 2
↳      //
103:         // a = amp param * n
↳      //
104:         // b = D + a.(S - D)
↳      //
105:         // D = invariant
↳      //
106:         // S = sum of balances but x,y = 0 since x  and y are the only
↳      tokens      //
107:         *****/
↳      *****/
108:
109:         unchecked {
110:             uint256 a = (amplificationParameter * 2) / _AMP_PRECISION;
111:             uint256 b = Math.mul(invariant, a).sub(invariant);
112:
113:             uint256 axy2 = Math.mul(a * 2, balanceX).mulDown(balanceY); //
↳      n = 2
114:
115:             // dx = a.x.y.2 + a.y^2 - b.y
116:             uint256 derivativeX = axy2.add(Math.mul(a,
↳      balanceY).mulDown(balanceY)).sub(b.mulDown(balanceY));
117:
118:             // dy = a.x.y.2 + a.x^2 - b.x
119:             uint256 derivativeY = axy2.add(Math.mul(a,
↳      balanceX).mulDown(balanceX)).sub(b.mulDown(balanceX));

```



```

120:
121:         // The rounding direction is irrelevant as we're about to
    ↪ introduce a much larger error when converting to log
122:         // space. We use `divUp` as it prevents the result from being
    ↪ zero, which would make the logarithm revert. A
123:         // result of zero is therefore only possible with zero
    ↪ balances, which are prevented via other means.
124:         return derivativeX.divUp(derivativeY);
125:     }
126: }

```

On a high level, the spot price is computed by determining the pool derivatives. The Balancer SDK's provide a feature to compute the spot price of any two tokens within a pool, and it leverages the \_poolDerivatives function.

The existing function for computing the spot price of any two tokens of a composable pool has the following errors or discrepancies from the approach used to compute the spot price in Balancer SDK, which might lead to an inaccurate spot price being computed.

#### Instance 1

The comments and SDK add  $b.y$  and  $b.x$  to the numerator and denominator, respectively, in the formula. However, the code performs a subtraction.

#### Instance 2

Per the comment and SDK code,  $b = (S - D)a + D$ .

However, assuming that  $S$  is zero (for a two-token pool), the following code in the Leverage Vault to compute  $b$  is not equivalent to the above.

```

uint256 b = Math.mul(invariant, a).sub(invariant);

```

#### Instance 3

The  $S$  in the code will always be zero because the code is catered only for two-token pools. However, for a composable pool, it can support up to five (5) tokens in a pool.  $S$  should be as follows, where *balances* is all the tokens in a composable pool except for BPT.

$$S = \sum_{i \neq \text{tokenIndexIn}, i \neq \text{tokenIndexOut}} \text{balances}[i]$$

#### Instance 4

The amplification factor is scaled by  $A * 2$  in the code, while the SDK scaled it by  $A * 2^2$  ( $A \text{TimesNpowN}$ ).





<https://github.com/balancer/balancer-sor/blob/73d6b435c1429bbfc199b39b38a36e581838d2c3/src/pools/stablePool/stableMath.ts#L235C63-L235C74>

### Instance 5

Per SDK, the amplification factor is scaled down by  $n^{(n-1)}$  where  $n$  is the number of tokens in a composable pool (excluding BPT). Otherwise, this was not implemented within the code.

### Impact

The spot price is used to verify if the pool has been manipulated before executing certain key vault actions (e.g. reinvest rewards). If the spot price is incorrect, it might potentially fail to detect the pool has been manipulated or result in unintended reverts due to false positives. In the worst-case scenario, the trade proceeds to execute against the manipulated pool, leading to a loss of assets.

### Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L93>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/math/StableMath.sol#L90>

### Tool used

Manual Review

### Recommendation

Given multiple discrepancies between the implementation of Leverage Vault's `_calcSpotPrice` function and SDK and due to the lack of information on the web, it is recommended to reach out to the Balancer's protocol team to identify the actual formula used to determine a spot price of any two tokens within a composable pool and check out if the formula in the SDK is up-to-date to be used against the composable pool.

It is also recommended to implement additional tests to ensure that the `_calcSpotPrice` returns the correct spot price of composable pools.

In addition, the `StableMath._calcSpotPrice` function is no longer used or found within the current version of Balancer's composable pool. Thus, there is no guarantee that the math within the `StableMath._calcSpotPrice` works with the current implementation. It is recommended to use the existing method in the current Composable Pool's `StableMath`, such as `_calcOutGivenIn` (ensure the fee is excluded) to compute the spot price.



## Discussion

**jeffywu**

Can confirm that this code no longer exists in the latest version of StableMath for the new ComposableStablePool, currently confirming with the Balancer team on how to approach this.

**jeffywu**

<https://github.com/notional-finance/leveraged-vaults/pull/63>

**xiaoming9090**

Fixed in [PR 63](#)



## Issue H-4: Fewer than expected LP tokens if the pool is imbalanced during vault restoration

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/82>

### Found by

mstpr-brainbot, xiaoming90

### Summary

The vault restoration function intends to perform a proportional deposit. If the pool is imbalanced due to unexpected circumstances, performing a proportional deposit is not optimal. This results in fewer pool tokens in return due to sub-optimal trade, eventually leading to a loss for the vault shareholder.

### Vulnerability Detail

Per the comment on Line 498, it was understood that the `restoreVault` function intends to deposit the withdrawn tokens back into the pool proportionally.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L501>

```
File: SingleSidedLPVaultBase.sol
498:    /// @notice Restores withdrawn tokens from emergencyExit back into the
    ↪ vault proportionally.
499:    /// Unlocks the vault after restoration so that normal functionality is
    ↪ restored.
500:    /// @param minPoolClaim slippage limit to prevent front running
501:    function restoreVault(
502:        uint256 minPoolClaim, bytes calldata /* data */
503:    ) external override whenLocked onlyNotionalOwner {
504:        StrategyVaultState memory state =
    ↪ VaultStorage.getStrategyVaultState();
505:
506:        (IERC20[] memory tokens, /* */ = TOKENS());
507:        uint256[] memory amounts = new uint256[](tokens.length);
508:
509:        // All balances held by the vault are assumed to be used to re-enter
510:        // the pool. Since the vault has been locked no other users should
    ↪ have
511:        // been able to enter the pool.
512:        for (uint256 i; i < tokens.length; i++) {
513:            if (address(tokens[i]) == address(POOL_TOKEN())) continue;
```



```

514:         amounts[i] = TokenUtils.tokenBalance(address(tokens[i]));
515:     }
516:
517:     // No trades are specified so this joins proportionally using the
518:     // amounts specified.
519:     uint256 poolTokens = _joinPoolAndStake(amounts, minPoolClaim);
    ..SNIP..

```

The main reason to join with all the pool's tokens in exact proportions is to minimize the price impact or slippage of the join. If the deposited tokens are imbalanced, they are often swapped internally within the pool, incurring slippage or fees.

However, the concept of proportional join to minimize slippage does not always hold with the current implementation of the `restoreVault` function.

### Proof-of-Concept

- 1) At T0, assume that a pool is perfectly balanced (50%-50%) with 1000 WETH and 1000 stETH.
- 2) At T1, an emergency exit is performed, the LP tokens are redeemed for the underlying pool tokens proportionally, and 100 WETH and 100 stETH are redeemed
- 3) At T2, certain events happen or due to ongoing issues with the pool (e.g., attacks, bugs, mass withdrawal), the pool becomes imbalanced (30%-70%) with 540 WETH and 1260 stETH.
- 4) At T3, the vault re-enters the withdrawn tokens to the pool proportionally with 100 WETH and 100 stETH. Since the pool is already imbalanced, attempting to enter the pool proportionally (50% WETH and 50% stETH) will incur additional slippage and penalties, resulting in fewer LP tokens returned.

This issue affects both Curve and Balancer pools since joining an imbalanced pool will always incur a loss.

### Explanation of imbalance pool

A Curve pool is considered imbalanced when there is an imbalance between the assets within it. For instance, the Curve stETH/ETH pool is considered imbalanced if it has the following reserves:

- ETH: 340,472.34 (31.70%)
- stETH: 733,655.65 (68.30%)

If a Curve Pool is imbalanced, attempting to perform a proportional join will not give an optimal return (e.g. result in fewer Pool LP tokens received).

In Curve Pool, there are penalties/bonuses when depositing to a pool. The pools



are always trying to balance themselves. If a deposit helps the pool to reach that desired balance, a deposit bonus will be given (receive extra tokens). On the other hand, if a deposit deviates from the pool from the desired balance, a deposit penalty will be applied (receive fewer tokens).

The following is the source code of `add_liquidity` function taken from <https://github.com/curvefi/curve-contract/blob/master/contracts/pools/steth/StableSwapSTETH.vy>. As shown below, the function attempts to calculate the difference between the `ideal_balance` and `new_balances`, and uses the difference as a factor of the fee computation, which is tied to the bonus and penalty.

```
def add_liquidity(amounts: uint256[N_COINS], min_mint_amount: uint256) ->
    uint256:
    ..SNIP..
    if token_supply > 0:
        # Only account for fees if we are not the first to deposit
        fee: uint256 = self.fee * N_COINS / (4 * (N_COINS - 1))
        admin_fee: uint256 = self.admin_fee
        for i in range(N_COINS):
            ideal_balance: uint256 = D1 * old_balances[i] / D0
            difference: uint256 = 0
            if ideal_balance > new_balances[i]:
                difference = ideal_balance - new_balances[i]
            else:
                difference = new_balances[i] - ideal_balance
            fees[i] = fee * difference / FEE_DENOMINATOR
            if admin_fee != 0:
                self.admin_balances[i] += fees[i] * admin_fee / FEE_DENOMINATOR
            new_balances[i] -= fees[i]
        D2 = self.get_D(new_balances, amp)
        mint_amount = token_supply * (D2 - D0) / D0
    else:
        mint_amount = D1 # Take the dust if there was any
    ..SNIP..
```

Following is the mathematical explanation of the penalties/bonuses extracted from Curve's Discord channel:

- There is a “natural” amount of D increase that corresponds to a given total deposit amount; when the pool is perfectly balanced, this D increase is optimally achieved by a balanced deposit. Any other deposit proportions for the same total amount will give you less D.
- However, when the pool is imbalanced, a balanced deposit is no longer optimal for the D increase.



## Impact

There is no guarantee that a pool will always be balanced. Historically, there have been multiple instances where the largest curve pool (stETH/ETH) has become imbalanced (Reference [#1](#) and [#2](#)).

If the pool is imbalanced due to unexpected circumstances, performing a proportional deposit is not optimal, leading to the deposit resulting in fewer LP tokens than possible due to the deposit penalty or slippage due to internal swap.

The side-effect is that the vault restoration will result in fewer pool tokens in return due to sub-optimal trade, eventually leading to a loss of assets for the vault shareholder.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L501>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L48>

## Tool used

Manual Review

## Recommendation

Consider providing the callers the option to deposit the reward tokens in a "non-proportional" manner if a pool becomes imbalanced. For instance, the function could allow the caller to swap the withdrawn tokens in external DEXs within the `restoreVault` function to achieve the most optimal proportion to minimize the penalty and slippage when re-entering the pool. This is similar to the approach in the vault's `reinvest` function.

## Discussion

**jeffyu**

This is a valid suggestion.

**jeffyu**

<https://github.com/notional-finance/leveraged-vaults/pull/75>

NOTE: this includes fixes from #82 to minimize conflicts.

**MLON33**



Fix: <https://github.com/notional-finance/leveraged-vaults/pull/75>

**xiaoming9090**

Fixed in [PR 75](#)



## Issue H-5: Incorrect invariant used for Balancer's composable pools

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/83>

### Found by

xiaoming90

### Summary

Only two balances instead of all balances were used when computing the invariant for Balancer's composable pools, which is incorrect. As a result, pool manipulation might not be detected. This could lead to the transaction being executed on the manipulated pool, resulting in a loss of assets.

### Vulnerability Detail

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L90>

```
File: BalancerSpotPrice.sol
78:     function _calculateStableMathSpotPrice(
    ..SNIP..
86:         // Apply scale factors
87:         uint256 secondary = balances[index2] * scalingFactors[index2] /
    ↪ BALANCER_PRECISION;
88:
89:         uint256 invariant = StableMath._calculateInvariant(
90:             ampParam, StableMath._balances(scaledPrimary, secondary), true
    ↪ // round up
91:         );
92:
93:         spotPrice = StableMath._calcSpotPrice(ampParam, invariant,
    ↪ scaledPrimary, secondary);
    ..SNIP..
```

A composable pool can support up to 5 tokens (excluding the BPT). When computing the invariant for a composable pool, one needs to pass in the balances of all the tokens within the pool except for BPT. However, the existing code always only passes in the balance of two tokens, which will return an incorrect invariant if the composable pool supports more than two tokens.

Following is the formula for computing the invariant of a composable pool taken from Balancer's Composable Pool. The `balances` passed into this function consist





of all balances except for BPT ([Reference](#))

<https://github.com/balancer/balancer-v2-monorepo/blob/c7d4abbea39834e7778f9ff7999aaceb4e8aa048/pkg/pool-stable/contracts/StableMath.sol#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
    //
    *****/
    *****/
```

The Balancer SDK's provide a feature to compute the spot price of any two tokens within a pool (<https://github.com/balancer/balancer-sdk/blob/develop/balancer-js/src/modules/pools/pool-types/concerns/stablePhantom/spotPrice.spec.ts>). By tracing the functions, it eventually triggers the following `_poolDerivatives` function.

Within the `_poolDerivatives` function, the balances used to compute the invariant consist of the balance of all tokens in the pool, except for BPT, which is aligned with the earlier understanding.

<https://github.com/balancer/balancer-sor/blob/73d6b435c1429bbfc199b39b38a36e581838d2c3/src/pools/phantomStablePool/phantomStableMath.ts#L516>

```
export function _poolDerivatives(
    A: BigNumber,
    balances: OldBigNumber[],
    tokenIndexIn: number,
    tokenIndexOut: number,
    is_first_derivative: boolean,
    wrt_out: boolean
```



```
) : OldBigNumber {  
    const totalCoins = balances.length;  
    const D = _invariant(A, balances);
```

Note: Composable Pool used to be called Phantom Pool in the past (<https://medium.com/balancer-protocol/rate-manipulation-in-balancer-boosted-pools-technical-postmortem-53db4b642492>)

## Impact

An incorrect invariant will lead to an incorrect spot price being computed. The spot price is used within the `_checkPriceAndCalculateValue` function that is intended to revert if the spot price on the pool is not within some deviation tolerance of the implied oracle price to prevent any pool manipulation. As a result, incorrect spot price leads to false positives or false negatives, where, in the worst-case scenario, pool manipulation was not caught by this function, and the transaction proceeded to be executed.

The `_checkPriceAndCalculateValue` function was found to be used within the following functions:

- `reinvestReward` - If the `_checkPriceAndCalculateValue` function is malfunctioning, it will cause the vault to add liquidity into a pool that has been manipulated, leading to a loss of assets.
- `convertStrategyToUnderlying` - This function is used by Notional V3 for the purpose of computing the collateral values and the account's health factor. If the `_checkPriceAndCalculateValue` function reverts unexpectedly due to an incorrect invariant/spot price, many of Notional's core functions will break. In addition, the collateral values and the account's health factor might be inflated if it fails to detect a manipulated pool due to incorrect invariant/spot price, potentially allowing the malicious actors to drain the main protocol.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/BalancerSpotPrice.sol#L90>

## Tool used

Manual Review

## Recommendation

Review if there is any specific reason for passing in only the balance of two tokens when computing the invariant. Otherwise, the balance of all tokens (except BPT)



should be used to compute the invariant.

In addition, it is recommended to include additional tests to ensure that the computed spot price is aligned with the market price.

## Discussion

**jeffyu**

This looks to be valid and we can re-align our implementation to match the balancer implementation.

**jeffyu**

<https://github.com/notional-finance/leveraged-vaults/pull/63>

**xiaoming9090**

Fixed in [PR 63](#)



## Issue H-6: Unable to reinvest if the reward token equals one of the pool tokens

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/84>

### Found by

coffiasd, mstpr-brainbot, xiaoming90

### Summary

If the reward token is the same as one of the pool tokens, the protocol would not be able to reinvest such a reward token. Thus leading to a loss of assets for the vault shareholders.

### Vulnerability Detail

During the reinvestment process, the `reinvestReward` function will be executed once for each reward token. The length of the `trades` listing defined in the payload must be the same as the number of tokens in the pool per Line 339 below.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L385>

```
File: SingleSidedLPVaultBase.sol
385:     function reinvestReward(
386:         SingleSidedRewardTradeParams[] calldata trades,
387:         uint256 minPoolClaim
388:     ) external whenNotLocked onlyRole(REWARD_REINVESTMENT_ROLE) returns (
389:         address rewardToken,
390:         uint256 amountSold,
391:         uint256 poolClaimAmount
392:     ) {
393:         // Will revert if spot prices are not in line with the oracle values
394:         _checkPriceAndCalculateValue();
395:
396:         // Require one trade per token, if we do not want to buy any tokens
397:         ↪ at a
398:         // given index then the amount should be set to zero. This applies
399:         ↪ to pool
400:         // tokens like in the ComposableStablePool.
401:         require(trades.length == NUM_TOKENS());
402:         uint256[] memory amounts;
403:         (rewardToken, amountSold, amounts) = _executeRewardTrades(trades);
```



In addition, due to the requirement at Line 105, each element in the `trades` listing must be a token within a pool and must be ordered in sequence according to the token index of the pool.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/StrategyUtils.sol#L90>

```
File: StrategyUtils.sol
090:     function executeRewardTrades(
091:         IERC20[] memory tokens,
092:         SingleSidedRewardTradeParams[] calldata trades,
093:         address rewardToken,
094:         address poolToken
095:     ) external returns(uint256[] memory amounts, uint256 amountSold) {
096:         amounts = new uint256[](trades.length);
097:         for (uint256 i; i < trades.length; i++) {
098:             // All trades must sell the same token.
099:             require(trades[i].sellToken == rewardToken);
100:             // Bypass certain invalid trades
101:             if (trades[i].amount == 0) continue;
102:             if (trades[i].buyToken == poolToken) continue;
103:
104:             // The reward trade can only purchase tokens that go into the
↳ pool
105:             require(trades[i].buyToken == address(tokens[i]));
```

Assuming the TriCRV Curve pool (crvUSD+WETH+CRV) has two reward tokens (CRV & CVX). This example is taken from a live Curve pool on Ethereum (Reference 1 Reference 2)

The pool will consist of the following tokens:

```
tokens[0] = crvUSD
tokens[1] = WETH
tokens[2] = CRV
```

Thus, if the protocol receives 3000 CVX reward tokens and it intends to sell 1000 CVX for crvUSD and 1000 CVX for WETH.

The trades list has to be defined as below.

```
trades[0].sellToken[0] = CRV (rewardToken) | trades[0].buyToken = crvUSD |
↳ trades[0].amount = 1000
trades[1].sellToken[1] = CRV (rewardToken) | trades[1].buyToken = WETH |
↳ trades[1].amount = 1000
```



```
trades[1].sellToken[2] = CRV (rewardToken) | trades[1].buyToken = CRV |
↳ trades[0].amount = 0
```

The same issue also affects the Balancer pools. Thus, the example is omitted for brevity. One of the affected Balancer pools is as follows, where the reward token is also one of the pool tokens.

- WETH-AURA - [Reference 1](#) [Reference 2](#) (Reward Tokens = [BAL, AURA])

However, the issue is that the `_isInvalidRewardToken` function within the `_executeRewardTrades` will always revert.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L413>

```
File: SingleSidedLPVaultBase.sol
413:     function _executeRewardTrades(SingleSidedRewardTradeParams[] calldata
↳ trades) internal returns (
414:         address rewardToken,
415:         uint256 amountSold,
416:         uint256[] memory amounts
417:     ) {
418:         // The sell token on all trades must be the same (checked inside
↳ executeRewardTrades) so
419:         // just validate here that the sellToken is a valid reward token
↳ (i.e. none of the tokens
420:         // used in the regular functioning of the vault).
421:         rewardToken = trades[0].sellToken;
422:         if (_isInvalidRewardToken(rewardToken)) revert
↳ Errors.InvalidRewardToken(rewardToken);
423:         (IERC20[] memory tokens, /* */) = TOKENS();
424:         (amounts, amountSold) = StrategyUtils.executeRewardTrades(
425:             tokens, trades, rewardToken, address(POOL_TOKEN())
426:         );
427:     }
```

The reason is that within the `_isInvalidRewardToken` function it checks if the reward token to be sold is any of the pool tokens. In this case, the condition will be evaluated to be true, and a revert will occur. As a result, the protocol would not be able to reinvest such reward tokens.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/mixins/AuraStakingMixin.sol#L38>

```
File: AuraStakingMixin.sol
38:     function _isInvalidRewardToken(address token) internal override view
↳ returns (bool) {
```



```

39:         return (
40:             token == TOKEN_1 ||
41:             token == TOKEN_2 ||
42:             token == TOKEN_3 ||
43:             token == TOKEN_4 ||
44:             token == TOKEN_5 ||
45:             token == address(AURA_BOOSTER) ||
46:             token == address(AURA_REWARD_POOL) ||
47:             token == address(Deployments.WETH)
48:         );
49:     }

```

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/curve/ConvexStakingMixin.sol#L60>

```

File: ConvexStakingMixin.sol
60:     function _isInvalidRewardToken(address token) internal override view
    ↪ returns (bool) {
61:         return (
62:             token == TOKEN_1 ||
63:             token == TOKEN_2 ||
64:             token == address(CURVE_POOL_TOKEN) ||
65:             token == address(CONVEX_REWARD_POOL) ||
66:             token == address(CONVEX_BOOSTER) ||
67:             token == Deployments.ALT_ETH_ADDRESS
68:         );
69:     }

```

## Impact

The reinvestment of reward tokens is a critical component of the vault. The value per vault share increases when reward tokens are sold for the pool tokens and reinvested back into the Curve/Balancer pool to obtain more LP tokens. If this feature does not work as intended, it will lead to a loss of assets for the vault shareholders.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L385>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/mixins/AuraStakingMixin.sol#L38>



<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/curve/ConvexStakingMixin.sol#L60>

## Tool used

Manual Review

## Recommendation

Consider tracking the number of pool tokens received during an emergency exit, and segregate these tokens with the reward tokens. For instance, the vault has 3000 CVX, 1000 of them are received during the emergency exit, while the rest are reward tokens emitted from Convex/Aura. In this case, the protocol can sell all CVX on the vault except for the 1000 CVX reserved.

## Discussion

**jeffywu**

Fair point, we will need to make some accommodations in these cases.

**jeffywu**

<https://github.com/notional-finance/leveraged-vaults/pull/73>

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/73>

**xiaoming9090**

Fixed in [PR 73](#)

The current implementation will work as long as there is no more than one reward token equal to one of the pool tokens. However, it is technically possible for a pool to hold more than one reward token. The existing implementation of the leverage vault will not be able to support such a pool. Given that the contracts are upgradable and such pools are rare, this limitation is of a lesser concern.

For your information, @jeffywu @T-Woodward

**jeffywu**

Acknowledged, however, in the interest of keeping things manageable I just 1 reward token on the whitelist is sufficient I think.





## Issue H-7: Different spot prices used during the comparison

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/85>

### Found by

xiaoming90

### Summary

The spot prices used during the comparison are different, which might result in the trade proceeding even if the pool is manipulated, leading to a loss of assets.

### Vulnerability Detail

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L90>

```
File: BalancerComposableAuraVault.sol
090:     function _checkPriceAndCalculateValue() internal view override returns
    ↪ (uint256) {
091:         (uint256[] memory balances, uint256[] memory spotPrices) =
    ↪ SPOT_PRICE.getComposableSpotPrices(
092:             BALANCER_POOL_ID,
093:             address(BALANCER_POOL_TOKEN),
094:             PRIMARY_INDEX()
095:         );
096:
097:         // Spot prices are returned in native decimals, convert them all to
    ↪ POOL_PRECISION
098:         // as required in the _calculateLPTokenValue method.
099:         (/ * /, uint8[] memory decimals) = TOKENS();
100:         for (uint256 i; i < spotPrices.length; i++) {
101:             spotPrices[i] = spotPrices[i] * POOL_PRECISION() / 10 **
    ↪ decimals[i];
102:         }
103:
104:         return _calculateLPTokenValue(balances, spotPrices);
105:     }
```

Line 91 above calls the `SPOT_PRICE.getComposableSpotPrices` function to fetch the spot prices. Within the function, it relies on the `StableMath._calcSpotPrice` function to compute the spot price. Per the comments of this function, spot price of token



Y in token X and spot price Y/X means that the Y (base) / X (quote). Thus, secondary (base) / primary (quote).

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/math/StableMath.sol#L90>

```
File: StableMath.sol
087:    /**
088:     * @dev Calculates the spot price of token Y in token X.
089:     */
090:    function _calcSpotPrice(
091:        uint256 amplificationParameter,
092:        uint256 invariant,
093:        uint256 balanceX,
094:        uint256 balanceY
095:    ) internal pure returns (uint256) {
096:        /*****
↳      *****/
097:        //
↳      //
098:        //      2.a.x.y + a.y^2 + b.y
↳      //
099:        // spot price Y/X = - dx/dy = -----
↳      //
100:        //      2.a.x.y + a.x^2 + b.x
↳      //
101:        //
↳      //
102:        // n = 2
↳      //
103:        // a = amp param * n
↳      //
104:        // b = D + a.(S - D)
↳      //
105:        // D = invariant
↳      //
106:        // S = sum of balances but x,y = 0 since x and y are the only
↳      tokens //
107:        *****/
↳      *****/
108:
109:        unchecked {
110:            uint256 a = (amplificationParameter * 2) / _AMP_PRECISION;
```

The above spot price will be used within the `_calculateLPTokenValue` function to compare with the oracle price to detect any potential pool manipulation. However,



the oracle price returned is in primary (base) / secondary (quote) format. As such, the comparison between the spot price (secondary-base/primary-quote) and oracle price (primary-base/secondary-quote) will be incorrect.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L355>

```
File: SingleSidedLPVaultBase.sol
333:     function _calculateLPTokenValue(
    ..SNIP..
351:         uint256 price = _getOraclePairPrice(primaryToken,
    ↪ address(tokens[i]));
352:
353:         // Check that the spot price and the oracle price are near
    ↪ each other. If this is
354:         // not true then we assume that the LP pool is being
    ↪ manipulated.
355:         uint256 lowerLimit = price * (Constants.VAULT_PERCENT_BASIS
    ↪ - limit) / Constants.VAULT_PERCENT_BASIS;
356:         uint256 upperLimit = price * (Constants.VAULT_PERCENT_BASIS
    ↪ + limit) / Constants.VAULT_PERCENT_BASIS;
357:         if (spotPrices[i] < lowerLimit || upperLimit <
    ↪ spotPrices[i]) {
358:             revert Errors.InvalidPrice(price, spotPrices[i]);
359:         }
```

## Impact

If the spot price is incorrect, it might potentially fail to detect the pool has been manipulated or result in unintended reverts due to false positives. In the worst-case scenario, the trade proceeds to execute against the manipulated pool, leading to a loss of assets.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L90>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/math/StableMath.sol#L90>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L355>



## Tool used

Manual Review

## Recommendation

Consider verifying if the comment of the `StableMath._calcSpotPrice` function is aligned with its implementation with the Balancer team.

In addition, the `StableMath._calcSpotPrice` function is no longer used or found within the current version of Balancer's composable pool. Thus, there is no guarantee that the math within the `StableMath._calcSpotPrice` works with the current implementation. It is recommended to use the existing method in the current Composable Pool's `StableMath`, such as `_calcOutGivenIn` (ensure the fee is excluded) to compute the spot price.

## Discussion

**jeffywu**

This fix is included in this PR:

<https://github.com/notional-finance/leveraged-vaults/pull/63>

**xiaoming9090**

Fixed in [PR 63](#)



## Issue H-8: Native ETH not received when removing liquidity from Curve V2 pools

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/86>

### Found by

xiaoming90

### Summary

Native ETH was not received when removing liquidity from Curve V2 pools due to the mishandling of Native ETH and WETH, leading to a loss of assets.

### Vulnerability Detail

Curve V2 pool will always wrap to WETH and send to leverage vault unless the `use_eth` is explicitly set to `True`. Otherwise, it will default to `False`. The following implementation of the `remove_liquidity_one_coin` function taken from one of the Curve V2 pools shows that unless the `use_eth` is set to `True`, the `WETH.deposit()` will be triggered to wrap the ETH, and WETH will be transferred back to the caller. The same is true for the `remove_liquidity` function, but it is omitted for brevity.

<https://etherscan.io/address/0x0f3159811670c117c372428d4e69ac32325e4d0f#code>

```
@external
@nonreentrant('lock')
def remove_liquidity_one_coin(token_amount: uint256, i: uint256, min_amount:
    ↪ uint256,
                                use_eth: bool = False, receiver: address =
                                ↪ msg.sender) -> uint256:
    A_gamma: uint256[2] = self._A_gamma()

    dy: uint256 = 0
    D: uint256 = 0
    p: uint256 = 0
    xp: uint256[N_COINS] = empty(uint256[N_COINS])
    future_A_gamma_time: uint256 = self.future_A_gamma_time
    dy, p, D, xp = self._calc_withdraw_one_coin(A_gamma, token_amount, i,
    ↪ (future_A_gamma_time > 0), True)
    assert dy >= min_amount, "Slippage"

    if block.timestamp >= future_A_gamma_time:
        self.future_A_gamma_time = 1
```



```

self.balances[i] -= dy
CurveToken(self.token).burnFrom(msg.sender, token_amount)

coin: address = self.coins[i]
if use_eth and coin == WETH20:
    raw_call(receiver, b"", value=dy)
else:
    if coin == WETH20:
        WETH(WETH20).deposit(value=dy)
    response: Bytes[32] = raw_call(
        coin,
        _abi_encode(receiver, dy,
            ↪ method_id=method_id("transfer(address,uint256)")),
        max_outsize=32,
    )
    if len(response) != 0:
        assert convert(response, bool)

```

Notional's Leverage Vault only works with Native ETH. It was found that the `remove_liquidity_one_coin` and `remove_liquidity` functions are executed without explicitly setting the `use_eth` parameter to `True`. Thus, WETH instead of Native ETH will be returned during remove liquidity. As a result, these WETH will not be accounted for in the vault and result in a loss of assets.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol#L83C17-L83C77>

```

File: Curve2TokenConvexVault.sol
66:     function _unstakeAndExitPool(
..SNIP..
78:         ICurve2TokenPool pool = ICurve2TokenPool(CURVE_POOL);
79:         exitBalances = new uint256[](2);
80:         if (isSingleSided) {
81:             // Redeem single-sided
82:             exitBalances[_PRIMARY_INDEX] = pool.remove_liquidity_one_coin(
83:                 poolClaim, int8(_PRIMARY_INDEX), _minAmounts[_PRIMARY_INDEX]
84:             );
85:         } else {
86:             // Redeem proportionally, min amounts are rewritten to a fixed
↪ length array
87:             uint256[2] memory minAmounts;
88:             minAmounts[0] = _minAmounts[0];
89:             minAmounts[1] = _minAmounts[1];
90:
91:             uint256[2] memory _exitBalances =
↪ pool.remove_liquidity(poolClaim, minAmounts);

```



```

92:         exitBalances[0] = _exitBalances[0];
93:         exitBalances[1] = _exitBalances[1];
94:     }

```

## Impact

Following are some of the impacts due to the mishandling of Native ETH and WETH during liquidity removal in Curve pools, leading to loss of assets:

1. Within the `redeemFromNotional`, if the vaults consist of ETH, the `_UNDERLYING_IS_ETH` will be set to true. In this case, the code will attempt to call `transfer` to transfer Native ETH, which will fail as Native ETH is not received and users/Notional are unable to redeem.

```

File: BaseStrategyVault.sol
175:     function redeemFromNotional(
..SNIP..
199:         if (_UNDERLYING_IS_ETH) {
200:             if (transferToReceiver > 0)
↳ payable(receiver).transfer(transferToReceiver);
201:             if (transferToNotional > 0)
↳ payable(address(NOTIONAL)).transfer(transferToNotional);
202:         } else {
..SNIP..

```

2. WETH will be received instead of Native ETH during the emergency exit. During vault restoration, WETH is not re-entered into the pool as only Native ETH residing in the vault will be transferred to the pool. Leverage vault only works with Native ETH, and if one of the pool tokens is WETH, it will be converted to Native ETH (0x0 or 0xEeeee) during deployment/initialization. Thus, the WETH is stuck in the vault. This causes the value per share to drop significantly. ([Reference](#))

```

File: SingleSidedLPVaultBase.sol
480:     function emergencyExit(
481:         uint256 claimToExit, bytes calldata /* data */
482:     ) external override onlyRole(EMERGENCY_EXIT_ROLE) {
483:         StrategyVaultState memory state =
↳ VaultStorage.getStrategyVaultState();
484:         if (claimToExit == 0) claimToExit = state.totalPoolClaim;
485:
486:         // By setting min amounts to zero, we will accept whatever
↳ tokens come from the pool
487:         // in a proportional exit. Front running will not have an
↳ effect since no trading will

```



```
488:         // occur during a proportional exit.
489:         _unstakeAndExitPool(claimToExit, new uint256[] (NUM_TOKENS()),
↳ true);
490:
491:         state.totalPoolClaim = state.totalPoolClaim - claimToExit;
492:         state.setStrategyVaultState();
```

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol#L83C17-L83C77>

## Tool used

Manual Review

## Recommendation

If one of the pool tokens is ETH, consider setting `is_eth` to true when calling `remove_liquidity_one_coin` and `remove_liquidity` functions to ensure that Native ETH is sent back to the vault.

## Discussion

**jeffyywu**

As I recall, this flag only exists on CurveV2 pools?

**jeffyywu**

<https://github.com/notional-finance/leveraged-vaults/pull/71>

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/71>

**xiaoming9090**

Fixed in [PR 71](#)





## Issue H-9: Single-sided instead of proportional exit is performed during emergency exit

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/87>

### Found by

xiaoming90

### Summary

Single-sided instead of proportional exit is performed during emergency exit, which could lead to a loss of assets during emergency exit and vault restoration.

### Vulnerability Detail

Per the comment in Line 476 below, the BPT should be redeemed proportionally to underlying tokens during an emergency exit. However, it was found that the `_unstakeAndExitPool` function is executed with the `isSingleSided` parameter set to `true`.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L480>

```
File: SingleSidedLPVaultBase.sol
475:     /// @notice Allows the emergency exit role to trigger an emergency exit
    ↪ on the vault.
476:     /// In this situation, the `claimToExit` is withdrawn proportionally to
    ↪ the underlying
477:     /// tokens and held on the vault. The vault is locked so that no
    ↪ entries, exits or
478:     /// valuations of vaultShares can be performed.
479:     /// @param claimToExit if this is set to zero, the entire pool claim is
    ↪ withdrawn
480:     function emergencyExit(
481:         uint256 claimToExit, bytes calldata /* data */
482:     ) external override onlyRole(EMERGENCY_EXIT_ROLE) {
483:         StrategyVaultState memory state =
    ↪ VaultStorage.getStrategyVaultState();
484:         if (claimToExit == 0) claimToExit = state.totalPoolClaim;
485:
486:         // By setting min amounts to zero, we will accept whatever tokens
    ↪ come from the pool
487:         // in a proportional exit. Front running will not have an effect
    ↪ since no trading will
```



```

488:         // occur during a proportional exit.
489:         _unstakeAndExitPool(claimToExit, new uint256[] (NUM_TOKENS()), true);

```

If the `isSingleSided` is set to `True`, the `EXACT_BPT_IN_FOR_ONE_TOKEN_OUT` will be used, which is incorrect. Per the Balancer's [documentation](#), `EXACT_BPT_IN_FOR_ONE_TOKEN_OUT` is a single asset exit where the user sends a precise quantity of BPT, and receives an estimated but unknown (computed at runtime) quantity of a single token.

To perform a proportional exit, the `EXACT_BPT_IN_FOR_TOKENS_OUT` should be used instead.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L67>

```

File: BalancerComposableAuraVault.sol
60:     function _unstakeAndExitPool(
61:         uint256 poolClaim, uint256[] memory minAmounts, bool isSingleSided
62:     ) internal override returns (uint256[] memory exitBalances) {
63:         bool success = AURA_REWARD_POOL.withdrawAndUnwrap(poolClaim, false);
64:         // claimRewards = false
65:         require(success);
66:         bytes memory customData;
67:         if (isSingleSided) {
68:             ..SNIP..
69:             uint256 primaryIndex = PRIMARY_INDEX();
70:             customData = abi.encode(
71:                 IBalancerVault.ComposableExitKind.EXACT_BPT_IN_FOR_ONE_TOKEN_OUT,
72:                 poolClaim,
73:                 primaryIndex < BPT_INDEX ? primaryIndex : primaryIndex - 1
74:             );

```

The same issue affects the Curve's implementation of the `_unstakeAndExitPool` function.

```

File: Curve2TokenConvexVault.sol
66:     function _unstakeAndExitPool(
67:         uint256 poolClaim, uint256[] memory _minAmounts, bool isSingleSided
68:     ) internal override returns (uint256[] memory exitBalances) {
69:         ..SNIP..
70:         ICurve2TokenPool pool = ICurve2TokenPool(CURVE_POOL);
71:         exitBalances = new uint256[] (2);
72:         if (isSingleSided) {
73:             // Redeem single-sided
74:             exitBalances[_PRIMARY_INDEX] = pool.remove_liquidity_one_coin(

```



```
83:                poolClaim, int8(_PRIMARY_INDEX), _minAmounts[_PRIMARY_INDEX]
84:            );
```

## Impact

The following are some of the impacts of this issue, which lead to loss of assets:

1. Redeeming LP tokens one-sided incurs unnecessary slippage as tokens have to be swapped internally to one specific token within the pool, resulting in fewer assets received.
2. Per the source code comment below, in other words, unless a proportional exit is performed, the emergency exit will be subjected to front-run attack and slippage.

```
File: SingleSidedLPVaultBase.sol
486:        // By setting min amounts to zero, we will accept whatever
    ↪ tokens come from the pool
487:        // in a proportional exit. Front running will not have an
    ↪ effect since no trading will
488:        // occur during a proportional exit.
489:        _unstakeAndExitPool(claimToExit, new uint256[] (NUM_TOKENS()),
    ↪ true);
```

3. After the emergency exit, the vault only held one of the pool tokens. To re-enter the pool, the vault has to either swap the token to other pool tokens on external DEXs to maintain the proportion or perform a single-sided join. Both of these methods will incur unnecessary slippage, resulting in fewer LP tokens received at the end.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L480>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerComposableAuraVault.sol#L67>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol#L80>

## Tool used

Manual Review



## Recommendation

Set the `isSingleSided` parameter to `false` when calling the `_unstakeAndExitPool` function to ensure that the proportional exit is performed.

```
function emergencyExit(
    uint256 claimToExit, bytes calldata /* data */
) external override onlyRole(EMERGENCY_EXIT_ROLE) {
    StrategyVaultState memory state = VaultStorage.getStrategyVaultState();
    if (claimToExit == 0) claimToExit = state.totalPoolClaim;
    ..SNIP..
-   _unstakeAndExitPool(claimToExit, new uint256[] (NUM_TOKENS()), true);
+   _unstakeAndExitPool(claimToExit, new uint256[] (NUM_TOKENS()), false);
```

## Discussion

**jeffyywu**

Valid issue

**jeffyywu**

<https://github.com/notional-finance/leveraged-vaults/pull/74>

Also includes a related fix for #80, in the changes above we do not use the token balances but owner will pass in the total amounts to re-enter the pools with manually.

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/74>

**xiaoming9090**

Fixed in [PR 74](#)



## Issue M-1: No check for active L2 Sequencer

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/2>

### Found by

0xMaroutis, Vagner, ZanyBonzy

### Summary

Using Chainlink in L2 chains such as Arbitrum requires to check if the sequencer is down to avoid prices from looking like they are fresh although they are not according to their recommendation

### Vulnerability Detail

The `SingleSidedLPVaultBase` and `CrossCurrencyVault` contracts make the `getOraclePrice` external call to the `TradingModule` contract. However, the `getOraclePrice` in the `TradingModule` makes no check to see if the sequencer is down.

### Impact

If the sequencer goes down, the protocol will allow users to continue to operate at the previous (stale) rates and this can be leveraged by malicious actors to gain unfair advantage.

### Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/7aadd254da5f645a7e1b718e7f9128f845e10f02/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L323>

```
function _getOraclePairPrice(address base, address quote) internal view returns
    ↪ (uint256) {
    (int256 rate, int256 precision) = TRADING_MODULE.getOraclePrice(base, quote);
    require(rate > 0);
    require(precision > 0);
    return uint256(rate) * POOL_PRECISION() / uint256(precision);
}
```

<https://github.com/sherlock-audit/2023-10-notional/blob/7aadd254da5f645a7e1b718e7f9128f845e10f02/leveraged-vaults/contracts/vaults/CrossCurrencyVault.sol#L131>



```
(int256 rate, int256 rateDecimals) = TRADING_MODULE.getOraclePrice(
```

<https://github.com/sherlock-audit/2023-10-notional/blob/7aadd254da5f645a7e1b718e7f9128f845e10f02/leveraged-vaults/contracts/trading/TradingModule.sol#L71C1-L77C6>

```
function getOraclePrice(address baseToken, address quoteToken)
    public
    view
    override
    returns (int256 answer, int256 decimals)
{
    PriceOracle memory baseOracle = priceOracles[baseToken];
    PriceOracle memory quoteOracle = priceOracles[quoteToken];

    int256 baseDecimals = int256(10**baseOracle.rateDecimals);
    int256 quoteDecimals = int256(10**quoteOracle.rateDecimals);

    (/* */, int256 basePrice, /* */, uint256 bpUpdatedAt, /* */) =
↳ baseOracle.oracle.latestRoundData();
    require(block.timestamp - bpUpdatedAt <= maxOracleFreshnessInSeconds);
    require(basePrice > 0); /// @dev: Chainlink Rate Error

    (/* */, int256 quotePrice, /* */, uint256 qpUpdatedAt, /* */) =
↳ quoteOracle.oracle.latestRoundData();
    require(block.timestamp - qpUpdatedAt <= maxOracleFreshnessInSeconds);
    require(quotePrice > 0); /// @dev: Chainlink Rate Error

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

## Tool used

Manual Review

## Recommendation

It is recommended to follow the Chailink [example code](#)



## Discussion

**jeffyu**

Valid, good suggestion

**jeffyu**

<https://github.com/notional-finance/leveraged-vaults/pull/76>

**nevillehuang**

Just to further avoid any potential escalations (if it helps), I am aware of the recent sherlock rule changes [here](#):

20. Chain re-org and network liveness related issues are not considered valid. Exception: If an issue concerns any kind of a network admin (e.g. a sequencer), can be remedied by a smart contract modification, the procol team considers external admins restricted and the considered network was explicitly mentioned in the contest README, it may be a valid medium. It should be assumed that any such network issues will be resolved within 7 days, if that may be possible.

Imo, this constitutes a valid medium given considered network (Arbitrum) was explicitly mentioned in the contest README and external admins are restricted as seen below here

On what chains are the smart contracts going to be deployed?

Arbitrum, Mainnet, Optimism

and here:

Are the admins of the protocols your contracts integrate with (if any) TRUSTED or RESTRICTED?

RESTRICTED, see answer to question below: "In case of external protocol integrations, are the risks of external contracts pausing or executing an emergency withdrawal acceptable?" Our understanding of the external protocols is that the scope of admin functionality is restricted.

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/76>

**xiaoming9090**

Fixed in [PR 76](#)



## Issue M-2: reinvestReward() generates dust totalPoolClaim causing vault abnormal

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/17>

### Found by

bin2chen, tvdung94

### Summary

If the first user deposits too small, due to the round down, it may result in 0 shares, which will result in 0 shares no matter how much is deposited later. In Notional, this situation will be prevented by setting a minimum borrow size and a minimum leverage ratio. However, `reinvestReward()` does not have this restriction, which may cause this problem to still exist, causing the vault to enter an abnormal state.

### Vulnerability Detail

The calculation of the shares of the vault is as follows:

```
function _mintVaultShares(uint256 lpTokens) internal returns (uint256
↳ vaultShares) {
    StrategyVaultState memory state = VaultStorage.getStrategyVaultState();
    if (state.totalPoolClaim == 0) {
        // Vault Shares are in 8 decimal precision
    @> vaultShares = (lpTokens *
↳ uint256(Constants.INTERNAL_TOKEN_PRECISION)) / POOL_PRECISION();
    } else {
        vaultShares = (lpTokens * state.totalVaultSharesGlobal) /
↳ state.totalPoolClaim;
    }

    // Updates internal storage here
    state.totalPoolClaim += lpTokens;
    state.totalVaultSharesGlobal += vaultShares.toUint80();
    state.setStrategyVaultState();
}
```

If the first deposit is too small, due to the conversion to `INTERNAL_TOKEN_PRECISION`, the precision is lost, resulting in `vaultShares=0`. Subsequent depositors will enter the second calculation, but `totalVaultSharesGlobal=0`, so `vaultShares` will always be 0.

To avoid this situation, Notional has restrictions.





hey guys, just to clarify some rounding issues stuff on vault shares and the precision loss. Notional will enforce a minimum borrow size and a minimum leverage ratio on users which will essentially force their initial deposits to be in excess of any dust amount. so we should not really see any tiny deposits that result in rounding down to zero vault shares. If there was rounding down to zero, the account will likely fail their collateral check as the vault shares act as collateral and the would have none. there is the possibility of a dust amount entering into depositFromNotional in a valid state, that would be due to an account "rolling" a position from one debt maturity to another. in this case, a small excess amount of deposit may come into the vault but the account would still be forced to be holding a sizeable position overall due to the minium borrow size.

in reinvestReward(), not this limit

```
function reinvestReward(
    SingleSidedRewardTradeParams[] calldata trades,
    uint256 minPoolClaim
) external whenNotLocked onlyRole(REWARD_REINVESTMENT_ROLE) returns (
    address rewardToken,
    uint256 amountSold,
    uint256 poolClaimAmount
) {
    // Will revert if spot prices are not in line with the oracle values
    _checkPriceAndCalculateValue();

    // Require one trade per token, if we do not want to buy any tokens at a
    // given index then the amount should be set to zero. This applies to
    ↪ pool
    // tokens like in the ComposableStablePool.
    require(trades.length == NUM_TOKENS());
    uint256[] memory amounts;
    (rewardToken, amountSold, amounts) = _executeRewardTrades(trades);

    poolClaimAmount = _joinPoolAndStake(amounts, minPoolClaim);

    // Increase LP token amount without minting additional vault shares
    StrategyVaultState memory state = VaultStorage.getStrategyVaultState();
    @> state.totalPoolClaim += poolClaimAmount;
    state.setStrategyVaultState();

    emit RewardReinvested(rewardToken, amountSold, poolClaimAmount);
}
```

From the above code, we know that reinvestReward() will increase totalPoolClaim,



but will not increase `totalVaultSharesGlobal`.

This will cause problems in the following scenarios:

1. The current vault has deposits.
2. Rewards have been generated, but `reinvestReward()` has not been executed.
3. The bot submitted the `reinvestReward()` transaction. but step 4 execute first
4. The users took away all the deposits `totalPoolClaim = 0`,  
`totalVaultSharesGlobal=0`.
5. At this time `reinvestReward()` is executed, then `totalPoolClaim > 0`,  
`totalVaultSharesGlobal=0`.
6. Other users' deposits will fail later

It is recommended that `reinvestReward()` add a judgment of `totalVaultSharesGlobal>0`.

Note: If there is a malicious `REWARD_REINVESTMENT_ROLE`, it can provoke this issue by donating reward token and triggering `reinvestReward()` before the first depositor appears.

## Impact

`reinvestReward()` generates dust `totalPoolClaim` causing vault abnormal

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L385>

## Tool used

Manual Review

## Recommendation

```
function reinvestReward(
    SingleSidedRewardTradeParams[] calldata trades,
    uint256 minPoolClaim
) external whenNotLocked onlyRole(REWARD_REINVESTMENT_ROLE) returns (
    address rewardToken,
    uint256 amountSold,
    uint256 poolClaimAmount
) {
```



```

        // Will revert if spot prices are not in line with the oracle values
        _checkPriceAndCalculateValue();

        // Require one trade per token, if we do not want to buy any tokens at a
        // given index then the amount should be set to zero. This applies to
    ↪ pool
        // tokens like in the ComposableStablePool.
        require(trades.length == NUM_TOKENS());
        uint256[] memory amounts;
        (rewardToken, amountSold, amounts) = _executeRewardTrades(trades);

        poolClaimAmount = _joinPoolAndStake(amounts, minPoolClaim);

        // Increase LP token amount without minting additional vault shares
        StrategyVaultState memory state = VaultStorage.getStrategyVaultState();
    + require(state.totalVaultSharesGlobal > 0 , "invalid shares");
        state.totalPoolClaim += poolClaimAmount;
        state.setStrategyVaultState();

        emit RewardReinvested(rewardToken, amountSold, poolClaimAmount);
    }

```

## Discussion

**jeffywu**

I'll mark this as valid because it is a good require check to have, although I think the reasoning on how this might arise is flawed. This would arise if an account enters the vault and then exits before the reinvestment is made.

**jeffywu**

<https://github.com/notional-finance/leveraged-vaults/pull/68>

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/68>

**xiaoming9090**

Fixed in [PR 68](#)



## Issue M-3: BalancerWeightedAuraVault.sol wrongly assumes that all of the weighted pools uses totalSupply

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/36>

### Found by

Vagner

### Summary

BalancerWeightedAuraVault.sol which is used specifically for weighted balancer pools wrongly uses totalSupply all the time to get the total supply of the pool, but this would not be true for newer weighted pools.

### Vulnerability Detail

Balancer pools have different methods to get their total supply of minted LP tokens, which is also specified in the docs here <https://docs.balancer.fi/concepts/advanced/valuing-bpt/valuing-bpt.html#getting-bpt-supply>. The docs specifies the fact that totalSupply is only used for older stable and weighted pools, and should not be used without checking it first, since the newer pools have pre-minted BPT and getActualSupply should be used in that case. Most of the time, the assumption would be that only the new composable stable pools uses the getActualSupply, but that is not the case, since even the newer weighted pools have and uses the getActualSupply. To give you few examples of newer weighted pools that uses getActualSupply <https://etherscan.io/address/0x9f9d900462492d4c21e9523ca95a7cd86142f298> <https://etherscan.io/address/0x3ff3a210e57cfe679d9ad1e9ba6453a716c56a2e> <https://etherscan.io/address/0xcf7b51ce5755513d4be016b0e28d6edeffa1d52a> the last one also being on Aura finance. Because of that, the interaction with newer weighted pools and also the future weighted pools would not be accurate since the wrong total supply is used and calculations like \_mintVaultShares and \_calculateLPTokenValue would both be inaccurate, which would hurt the protocol and the users.

### Impact

Impact is a high one since the calculation of shares and the value of the LP tokens is very important for the protocol, and any miscalculations could hurt the protocol and the users a lot.



## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/BalancerWeightedAuraVault.sol#L1-L94>

## Tool used

Manual Review

## Recommendation

Since the protocol would want to interact with multiple weighted pools, try to check first if the weighted pool you are interacting with is a newer one and uses the `getActualSupply` or if it is an older one and uses `totalSupply`, in that way the protocol could interact with multiple pools in the long run.

## Discussion

**jeffyu**

Thanks, good finding.

**jeffyu**

<https://github.com/notional-finance/leveraged-vaults/pull/69>

**gstoyanovbg**

As I read the report, I left with the impression that the `getActualSupply` function should be used because there is some amount of pre-minted BPT tokens. However, this is not true in the case of weighted pools. In those cases, we don't have pre-minted tokens, and consequently, there is no virtual supply. This is evident from the links to the contracts provided in the description. This is the code of the `getActualSupply` function.

```
/**
 * @notice Returns the effective BPT supply.
 *
 * @dev This would be the same as `totalSupply` however the Pool owes debt to
 ↳ the Protocol in the form of unminted
 * BPT, which will be minted immediately before the next join or exit. We need
 ↳ to take these into account since,
 * even if they don't yet exist, they will effectively be included in any Pool
 ↳ operation that involves BPT.
 *
 * In the vast majority of cases, this function should be used instead of
 ↳ `totalSupply()`.
 */
```



```

* **IMPORTANT NOTE**: calling this function within a Vault context (i.e. in the
↳ middle of a join or an exit) is
* potentially unsafe, since the returned value is manipulable. It is up to the
↳ caller to ensure safety.
*
* This is because this function calculates the invariant, which requires the
↳ state of the pool to be in sync
* with the state of the Vault. That condition may not be true in the middle of
↳ a join or an exit.
*
* To call this function safely, attempt to trigger the reentrancy guard in the
↳ Vault by calling a non-reentrant
* function before calling `getActualSupply`. That will make the transaction
↳ revert in an unsafe context.
* (See `whenNotInVaultContext` in `WeightedPool`).
*
* See https://forum.balancer.fi/t/reentrancy-vulnerability-scope-expanded/4345
↳ for reference.
*/
function getActualSupply() external view returns (uint256) {
    uint256 supply = totalSupply();

    (uint256 protocolFeesToBeMinted, ) = _getPreJoinExitProtocolFees(
        getInvariant(),
        _getNormalizedWeights(),
        supply
    );

    return supply.add(protocolFeesToBeMinted);
}

```

From the comment, it is apparent that this function should be used because the pool owes debt to the Protocol in the form of unminted BPT, which should be taken into consideration. Despite the incorrect reason, I agree that `getActualSupply` should be used instead of `totalSupply`. I'm not entirely sure about the impact, and I won't comment on it. I'm writing this comment because the report will be read by people after the contest, and they would receive incorrect information.

**xiaoming9090**

Reviewed [PR 69](#) and observed that the wrong interface is used. The `IWeightedPool` interface should be used instead of `IComposablePool` interface since this is a weighted pool vault.

**jeffyu**

@xiaoming9090 made a fix here:



<https://github.com/notional-finance/leveraged-vaults/pull/82>

Although the method signature is the same for both contracts so this doesn't make any real functional change. Agree that it is more clear this way.

**xiaoming9090**

Fixed in PR 82

**jeffyu**

@gstoyanovbg Yes your comment is correct. We followed up with the Balancer team directly on this. Newer Weighted Pools have this method exposed when fees are paid in BPT inflation. We would prefer to use this method over `totalSupply` to ensure that we have the correct denominator for the total BPT. I believe the two numbers are "trued up" when new LP tokens are minted so they should not diverge excessively over time.



## Issue M-4: Some curve pools can not be used as a single sided strategy

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/40>

### Found by

mstpr-brainbot

### Summary

For single sided curve strategy contest docs says that any curve pool should be ok to be used. However, some pools are not adaptable to the Curve2TokenPoolMixin abstract contract.

### Vulnerability Detail

The contract assumes the existence of three types of pool scenarios. In one scenario, the pool address itself serves as the LP token. In another scenario, the LP token is obtained by querying the lp\_token() or token() function. However, in some cases where potential integration with Notional is possible, certain pools lack a direct method to retrieve the underlying LP token. For instance, the sETH-ETH pool, which presents a promising option for a single-sided strategy in ETH vaults, does not offer public variables to access the underlying LP token. Although the pool contract contains the token variable that returns the LP token of the pool, this variable is not publicly accessible. Consequently, in such cases, querying the LP token directly from the pool is not feasible, and it becomes necessary to provide the LP token address as input.

Here the example contracts where neither of the options are available: sETH-ETH: <https://etherscan.io/address/0xc5424b857f758e906013f3555dad202e4bdb4567>  
hBTC-WBTC: <https://etherscan.io/address/0x4ca9b3063ec5866a4b82e437059d2c43d1be596f>

### Impact

All curve pools that can be used as a single sided strategy for notional leveraged vaults considered to be used but some pools are not adaptable thus I will label this as medium.

### Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/7aadd254da5f645a7e1b718e7f9128f845e10f02/leveraged-vaults/contracts/vaults/curve/Curve2TokenPool>





Mixin.sol#L73-L78

## Tool used

Manual Review

## Recommendation

Curve contracts are pretty different and it is very hard to make a generic template for it. I would suggest make the LP token also an input and remove the necessary code for setting it in the constructor. Since the owner is trusted this should not be a problem.

## Discussion

**jeffywu**

Valid suggestion.

**jeffywu**

<https://github.com/notional-finance/leveraged-vaults/pull/70>

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/70>

**xiaoming9090**

Fixed in [PR 70](#)



## Issue M-5: `depositFromNotional` function is payable, which means that it should accept Ether, but in reality will revert 100% when `msg.value > 0`

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/51>

### Found by

AuditorPraise, Vagner

### Summary

The function `depositFromNotional` used in `BaseStrategyVault.sol` is payable, which means that it should accept Ether, but in reality it will revert every time when `msg.value` is `>` than 0 in any of existing strategy.

### Vulnerability Detail

`depositFromNotional` is a function used in `BaseStrategyVault.sol` for every strategy, to deposit from notional to a specific strategy. As you can see this function has the `payable` keyword <https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/BaseStrategyVault.sol#L166-L173> which means that it is expected to be used along with `msg.value` being `>` than 0. This function would call `_depositFromNotional` which is different on any strategy used, but let's take the most simple case, since all of them will be the same in the end, the case of `CrossCurrencyVault.sol`. In `CrossCurrencyVault.sol`, `_depositFromNotional` would later call `_executeTrade` <https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/CrossCurrencyVault.sol#L184> which would use the `TradeHandler` library as can be seen here <https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/common/BaseStrategyVault.sol#L124> If we look into the `TradeHandler` library, `_executeTrade` would delegatecall into the implementation of `TradingModule.sol` to `executeTrade` function, as can be seen here <https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/trading/TradeHandler.sol#L41-L42> If we look into the `executeTrade` function in `TradingModule.sol` we can see that this function does not have the `payable` keyword, which means that it will not accept `msg.value >` than 0 <https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/trading/TradingModule.sol#L169-L193> The big and important thing to know about delegate call is that, `msg.sender` and `msg.value` will always be kept when you delegate call, so the problem that arises here is the fact that, the calls made to `depositFromNotional` with `msg.value > 0`, would always revert when it gets to this delegatecall, in every strategy, since the function that it delegates to, doesn't have



the payable keyword, and since msg.value is always kept through the delegate calls, the call would just revert. This would be the case for all the strategies since they all use \_executeTrade or \_executeTradeWithDynamicSlippage in a way or another, so every payable function that would use any of these 2 functions from the TradeHandler.sol library would revert all the time, if msg.value is > 0.

## Impact

Impact is a high one, since strategy using pools that need to interact with Ether would be useless and the whole functionality would be broken.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/trading/TradeHandler.sol#L18-L45>

## Tool used

Manual Review

## Recommendation

There are multiple solutions to this, the easiest one is to make the function executeTrade in TradingModule.sol payable, so the delegate call would not revert when msg.value is greater than 0, or if you don't intend to use ether with depositFromNotional, remove the payable keyword. It is important to take special care in those delegate calls since they are used multiple times in the codebase, and could mess up functionalities when msg.value is intended to be used.

## Discussion

jeffyywu

Trade Handler is executed in a delegate call context, I don't really follow this issue.

VagnerAndrei26

Escalate I want to escalate this issue and explain it a bit better so it can be understood. The core of this issue stands at the base of how delegatecall works. The big difference between normal call and delegatecall is, as I said, msg.sender and msg.value are kept within the call. Here is the source code of EVM

```
File: core\vm\contract.go
134: func (c *Contract) AsDelegate() *Contract {
135:     // NOTE: caller must, at all times be a contract. It should never happen
136:     // that caller is something other than a Contract.
```



```

137:    parent := c.caller.(*Contract)
138:    c.CallerAddress = parent.CallerAddress
139:    c.value = parent.value
140:
141:    return c
142: }

```

basically what this means is that, anytime when you have a function that is payable, and implicitly has `msg.value > 0`, that `msg.value` will always be passed in a `delegatecall`. In the case of normal `call` you can choose if you want to pass some value by using `{ value: }`, after calling the function in a contract, but for `delegatecall` you can't choose, so what will happen is that, any time you do a `delegatecall` inside a function that has payable and where `msg.value` is `> 0`, that value is always passed in that `delegatecall` and if the function called does not have the payable keyword, the transaction will always revert, the same as when you are trying to do a simple `call` to a contract without having a `receive` function or payable keyword on a function. So in our case, because `_executeTrade` or `_executeTradeWithDynamicSlippage` does not have the payable keyword, any call on `depositFromNotional` with `msg.value > 0`, would revert all the time, making the vaults in the cases where ETH is used useless. Also issue #8 is not a duplicate of this, since it would fall more to this duplicate rule of sherlock which states

- In addition to this, there is a submission **D** which identifies the core issue but does not clearly describe the impact or an attack path. Then **D** is considered low.

since the report identifies the core issue, which is lacking the payable keyword but doesn't correctly identifies the reason behind that being an issue or the impact.

## sherlock-admin2

Escalate I want to escalate this issue and explain it a bit better so it can be understood. The core of this issue stands at the base of how `delegatecall` works. The big difference between normal `call` and `delegatecall` is, as I said, `msg.sender` and `msg.value` are kept within the call. Here is the source code of EVM

```

File: core\vm\contract.go
134: func (c *Contract) AsDelegate() *Contract {
135:    // NOTE: caller must, at all times be a contract. It should never
    ↪ happen
136:    // that caller is something other than a Contract.
137:    parent := c.caller.(*Contract)
138:    c.CallerAddress = parent.CallerAddress
139:    c.value = parent.value
140:
141:    return c
142: }

```



basically what this means is that, anytime when you have a function that is payable, and implicitly has `msg.value > 0`, that `msg.value` will always be passed in a `delegatecall`. In the case of normal `call` you can choose if you want to pass some value by using `{ value: }`, after calling the function in a contract, but for `delegatecall` you can't choose, so what will happen is that, any time you do a `delegatecall` inside a function that has payable and where `msg.value` is `> 0`, that value is always passed in that `delegatecall` and if the function called does not have the `payable` keyword, the transaction will always revert, the same as when you are trying to do a simple `call` to a contract without having a `receive` function or `payable` keyword on a function. So in our case, because `_executeTrade` or `_executeTradeWithDynamicSlippage` does not have the `payable` keyword, any call on `depositFromNotional` with `msg.value > 0`, would revert all the time, making the vaults in the cases where ETH is used useless. Also issue #8 is not a duplicate of this, since it would fall more to this duplicate rule of `sherlock` which states

- In addition to this, there is a submission **D** which identifies the core issue but does not clearly describe the impact or an attack path. Then **D** is considered low.

since the report identifies the core issue, which is lacking the `payable` keyword but doesn't correctly identifies the reason behind that being an issue or the impact.

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.

## AuditorPraise

Escalate I want to escalate this issue and explain it a bit better so it can be understood. The core of this issue stands at the base of how `delegatecall` works. The big difference between normal `call` and `delegatecall` is, as I said, `msg.sender` and `msg.value` are kept within the call. Here is the source code of EVM

```
File: core\vm\contract.go
134: func (c *Contract) AsDelegate() *Contract {
135: // NOTE: caller must, at all times be a contract. It should never
    ↪ happen
136: // that caller is something other than a Contract.
137: parent := c.caller.(*Contract)
138: c.CallerAddress = parent.CallerAddress
139: c.value = parent.value
140:
141: return c
```



```
142: }
```

basically what this means is that, anytime when you have a function that is payable, and implicitly has `msg.value > 0`, that `msg.value` will always be passed in a `delegatecall`. In the case of normal `call` you can choose if you want to pass some value by using `{ value: }`, after calling the function in a contract, but for `delegatecall` you can't choose, so what will happen is that, any time you do a `delegatecall` inside a function that has payable and where `msg.value` is `> 0`, that value is always passed in that `delegatecall` and if the function called does not have the `payable` keyword, the transaction will always revert, the same as when you are trying to do a simple `call` to a contract without having a `receive` function or `payable` keyword on a function. So in our case, because `_executeTrade` or `_executeTradeWithDynamicSlippage` does not have the `payable` keyword, any call on `depositFromNotional` with `msg.value > 0`, would revert all the time, making the vaults in the cases where ETH is used useless. Also issue #8 is not a duplicate of this, since it would fall more to this duplicate rule of sherlock which states since the report identifies the core issue, which is lacking the `payable` keyword but doesn't correctly identifies the reason behind that being an issue or the impact.

LoL, how is #8 a low? Because i didn't use the same exact words you used when describing your issue? #8 's summary: The vaults will be executing trades on external exchanges via `TradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()` and ETH could be among the tokens to trade for primary token BUT the `tradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()` lack the `payable` keyword.

Its vulnerability detail: `tradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()` won't be able to receive ETH (Whenever ETH is sell token) because they lack the `payable` keyword.

This can cause reverts in some of the key functions of the vaults like:

`depositFromNotional()` `redeemFromNotional()` `reinvestReward()`

And its impact: vaults will be unable to execute trades on external exchanges via the trading module whenever ETH is the sell Token'

How are they different? #8 's impact is clearly described and i don't see how it's different from your described impact because they're both implying the same thing Anyways thanks for escalating sir

**VagnerAndrei26**

Escalate I want to escalate this issue and explain it a bit better so it can be understood. The core of this issue stands at the base of how `delegatecall` works. The big difference between



normal call and delegatecall is , as I said , msg.sender and msg.value are kept within the call. Here is the source code of EVM

```
File: core\vm\contract.go
134: func (c *Contract) AsDelegate() *Contract {
135:     // NOTE: caller must, at all times be a contract. It should
    ↪ never happen
136:     // that caller is something other than a Contract.
137:     parent := c.caller.(*Contract)
138:     c.CallerAddress = parent.CallerAddress
139:     c.value = parent.value
140:
141:     return c
142: }
```

basically what this means is that, anytime when you have a function that is payable, and implicitly has msg.value > 0 , that msg.value will always be passed in a delegatecall. In the case of normal call you can choose if you want to pass some value by using { value: }, after calling the function in a contract, but for delegatecall you can't choose, so what will happen is that, any time you do a delegatecall inside a function that has payable and where msg.value is > 0, that value is always passed in that delegatecall and if the function called does not have the payable keyword, the transaction will always revert, the same as when you are trying to do a simple call to a contract without having a receive function or payable keyword on a function. So in our case, because \_executeTrade or \_executeTradeWithDynamicSlippage does not have the payable keyword, any call on depositFromNotional with msg.value > 0 , would revert all the time, making the vaults in the cases where ETH is used useless. Also issue #8 is not a duplicate of this, since it would fall more to this duplicate rule of sherlock which states since the report identifies the core issue, which is lacking the payable keyword but doesn't correctly identifies the reason behind that being an issue or the impact.

LoL, how is #6 a low? Because i didn't use the same exact words you used when describing your issue? #6 's summary: The vaults will be executing trades on external exchanges via TradingModule.executeTrade() and TradingModule.executeTradeWithDynamicSlippage() and ETH could be among the tokens to trade for primary token BUT the tradingModule.executeTrade() and





TradingModule.executeTradeWithDynamicSlippage() lack the payable keyword.

Its vulnerability detail: tradingModule.executeTrade() and TradingModule.executeTradeWithDynamicSlippage() won't be able to receive ETH (Whenever ETH is sell token) because they lack the payable keyword.

This can cause reverts in some of the key functions of the vaults like:

depositFromNotional() redeemFromNotional() reinvestReward()

And its impact: vaults will be unable to execute trades on external exchanges via the trading module whenever ETH is the sell Token'

How are they different? #6 's impact is clearly described and i don't see how it's different from your described impact because they're both implying the same thing Anyways thanks for escalating sir

Hey, I didn't say that it was a low personally, I said that it is not a duplicate of this, and should be considered as the duplicate rule stated, if it is kept as a duplicate. The big difference is that the report failed to find the real problem of this whole issue, the `delegatecall`. The main important aspect of this issue is the fact that those functions, that were also specified #8, are used in a `delegatecall` and not in a normal call, if it were done in a normal call, this would have not be a problem since you can select if you want to transfer `msg.value` with the call or not. Also the report speaks about `redeemFromNotional` and `reinvestReward` having the same issue, but that is not correct since they are not payable, so no `msg.value` is expected to be used when calling those, the only real function affected by this issue is `depositFromNotional`. Also in the report it is also talking about `TradingModule` not being able to receive ether which is not a problem again, since the main usage of `TradingModule` is to be used more as an implementation, where calls could be delegated to. That's the reasons behind my arguments, the only thing which the report accurate is the fact the those function lacks the `payable` keyword, but fails to explain the real problem behind, or why is that a problem in the current code.

## AuditorPraise

Escalate I want to escalate this issue and explain it a bit better so it can be understood. The core of this issue stands at the base of how `delegatecall` works. The big difference between normal call and `delegatecall` is , as I said , `msg.sender` and `msg.value` are kept within the call. Here is the source code of EVM

```
File: core\vm\contract.go
134: func (c *Contract) AsDelegate() *Contract {
```





```
135: // NOTE: caller must, at all times be a contract. It should
    ↪ never happen
136: // that caller is something other than a Contract.
137: parent := c.caller.(*Contract)
138: c.CallerAddress = parent.CallerAddress
139: c.value = parent.value
140:
141: return c
142: }
```

basically what this means is that, anytime when you have a function that is payable, and implicitly has `msg.value > 0`, that `msg.value` will always be passed in a `delegatecall`. In the case of normal `call` you can choose if you want to pass some value by using `{ value: }`, after calling the function in a contract, but for `delegatecall` you can't choose, so what will happen is that, any time you do a `delegatecall` inside a function that has payable and where `msg.value` is `> 0`, that value is always passed in that `delegatecall` and if the function called does not have the `payable` keyword, the transaction will always revert, the same as when you are trying to do a simple `call` to a contract without having a `receive` function or `payable` keyword on a function. So in our case, because `_executeTrade` or `_executeTradeWithDynamicSlippage` does not have the `payable` keyword, any call on `depositFromNotional` with `msg.value > 0`, would revert all the time, making the vaults in the cases where ETH is used useless. Also issue #8 is not a duplicate of this, since it would fall more to this duplicate rule of sherlock which states since the report identifies the core issue, which is lacking the `payable` keyword but doesn't correctly identifies the reason behind that being an issue or the impact.

LoL, how is #6 a low? Because i didn't use the same exact words you used when describing your issue? #6 's summary: The vaults will be executing trades on external exchanges via `TradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()` and ETH could be among the tokens to trade for primary token BUT the `tradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()` lack the `payable` keyword. Its vulnerability detail:



`tradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()` won't be able to receive ETH (Whenever ETH is sell token) because they lack the payable keyword. This can cause reverts in some of the key functions of the vaults like: `depositFromNotional()` `redeemFromNotional()` `reinvestReward()` And its impact: vaults will be unable to execute trades on external exchanges via the trading module whenever ETH is the sell Token' How are they different? #6 's impact is clearly described and i don't see how it's different from your described impact because they're both implying the same thing Anyways thanks for escalating sir

Hey, I didn't say that it was a low personally, I said that it is not a duplicate of this, and should be considered as the duplicate rule stated, if it is kept as a duplicate. The big difference is that the report failed to find the real problem of this whole issue, the `delegatecall`. The main important aspect of this issue is the fact that those functions, that were also specified #8, are used in a `delegatecall` and not in a normal call, if it were done in a normal call, this would have not be a problem since you can select if you want to transfer `msg.value` with the call or not. Also the report speaks about `redeemFromNotional` and `reinvestReward` having the same issue, but that is not correct since they are not payable, so no `msg.value` is expected to be used when calling those, the only real function affected by this issue is `depositFromNotional`. Also in the report it is also talking about `TradingModule` not being able to receive ether which is not a problem again, since the main usage of `TradingModule` is to be used more as an implementation, where calls could be delegated to. That's the reasons behind my arguments, the only thing which the report accurate is the fact the those function lacks the payable keyword, but fails to explain the real problem behind, or why is that a problem in the current code.

The `delegateCall` is not the main issue bro. The lack of a payable keyword on the `tradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()` is the main issue. Because pools that need to interact with Ether would be useless due to the missing payable keyword on those external functions that are supposed to interact with ETH(i.e, `tradingModule.executeTrade()` and `TradingModule.executeTradeWithDynamicSlippage()`) and the whole functionality would be broken.

`Msg.value` will always be > 0 whenever the token being transferred is ETH

**nevillehuang**

Hi @VagnerAndrei26 @AuditorPraise can any of you provide me a coded test file



proving the issue? Imo if it is 100% going to revert, should be easy enough to proof it, appreciate your assistance.

### VagnerAndrei26

Hi @VagnerAndrei26 @AuditorPraise can any of you provide me a coded test file proving the issue? Imo if it is 100% going to revert, should be easy enough to proof it, appreciate your assistance.

Yeah I can, I can show a simpler implementation of 2 contracts doing some delegatecalls, I am 100% sure on this issue, because I got paid for it multiple times already and it got confirmed in other contest too. Will provide the simple example later today, after I get home.

### AuditorPraise

Hello bro @nevillehuang, i just wrote a quick implementation of 2 contracts doing some delegatecalls on remix. just copy and paste on remix

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity =0.8.18;

/**
 * @hypothesis
 * @dev just for research.
 */
import "hardhat/console.sol";

contract ContractA {
    receive() external payable {}
    address public contractBAddress;

    event EtherTransferred(address indexed from, address indexed to, uint256
↳ amount);

    constructor(address _contractBAddress) {
        contractBAddress = _contractBAddress;
    }

    function delegateCallToContractB(uint256 amount) payable external {
        // Use delegatecall to execute the transferEther function in ContractB
        (bool success, ) = payable(contractBAddress).delegatecall(
            abi.encodeWithSignature("transferEther(address,uint256)",
↳ msg.sender, amount)
        );

        require(success, "Delegate call to ContractB failed");
    }
}
```



```

        console.log("trf was successful");
    }
}

contract ContractB {
    receive() external payable {}

    function transferEther(address recipient, uint256 amount)
    external {
        //@audit-issue the delegate call from contract A fails without the
        ↪ payable keyword here

    }
}

```

So this is supposed to show the issue we are talking about. when contractB.transferEther()'s function has the payable keyword the delegate call from contract A succeeds BUT when its not there it reverts.

The code above should revert, then add the payable keyword to contractB.transferEther() you'll see that contractA.delegateCallToContractB() succeeds.

Here's an article that may be of help with sending Ether on remix  
[How do you send Ether as a function to a contract using Remix?](#)

### shealtielanz

The 2 reports are duplicates IMO, as the root issues is stated in both of them, and their mitigations solves the issues well.

### nevillehuang

@jeffyuw Is there a foundry test revolving this issue that shows it will not revert if u pass in a msg.value? I am really surprised if this is true that a test would not have caught this.

Since the PoC provided is not protocol specific, I will have to double check before making any final comment.

### VagnerAndrei26

@jeffyuw Is there a foundry test revolving this issue that shows it will not revert if u pass in a msg.value? I am really surprised if this is true that a test would not have caught this.

Since the PoC provided is not protocol specific, I will have to double



check before making any final comment.

The example provided from @AuditorPraise is good, it is pretty similar to what I wanted to present. The bases of this issue is simple, if you have a `delegatecall` inside a payable function, the function which you `delegatecall` to needs to have payable also, otherwise it reverts anytime when `msg.value > 0`.

## Czar102

Awaiting a protocol-specific PoC @VagnerAndrei26 and a comment from @jeffywu.

## VagnerAndrei26

Awaiting a protocol-specific PoC @VagnerAndrei26 and a comment from @jeffywu.

Hey @Czar102 it was kinda hard to do it, because of the complex codebase and test, but here is the protocol specific POC. What I did was modifying this specific test <https://github.com/sherlock-audit/2023-10-notional-VagnerAndrei26/blob/eba16143f4d71d84affb69258bcf00c375628d3/leveraged-vaults/tests/BaseAcceptanceTest.sol#L173-L190> by setting the `isETH` to true firstly, and giving the ETH to the NOTIONAL address instead of the vault itself. After that I was calling the `depositFromNotional` with a specific value, which reverts all the time when it gets to `executeTrade`. The modifications would look like this

```
function enterVaultBypass(
    address account,
    uint256 depositAmount,
    uint256 maturity,
    bytes memory data
) internal virtual returns (uint256 vaultShares) {
    vm.prank(address(NOTIONAL));
    deal(address(NOTIONAL), depositAmount);
    vaultShares = vault.depositFromNotional{value : depositAmount}(account,
    ↪ depositAmount, maturity, data);
    totalVaultShares[maturity] += vaultShares;
    totalVaultSharesAllMaturities += vaultShares;
}
```

and here is the trace logs, as you can see it revert at `executeTrade`, as I expected and explained in the report

```
Running 1 test for
↪ tests/CrossCurrency/testCrossCurrencyVault.t.sol:TestCrossCurrency_ETH_WSTETH
[FAIL. Reason: TradeFailed(); counterexample:
↪ calldata=0xde6f757a00000000000000000000000000000000000000000000000000000000
↪ 0000000000000000000000000000000000000000000000000000000000000000
↪ args=[0, 0]] test_EnterVault(uint256,uint256) (runs: 0, : 0, ~: 0)
```



[illegible]

```

    [213] 0x2De2B1Eecf5bab0adD9147ebBb999395238d30a5::executeTrade(7, (0,
↳ 0x0000000000000000000000000000000000000000000000000000000000000000,
↳ 0x5979D7b546E38E414F7E9822514be443A4800529, 1000000000000000 [1e15], 0,
↳ 1699104713 [1.699e9],
↳ 0x0000000000000000000000000000000000000000000000000000000000000000,
↳ [delegatecall]
    ← EvmError: Revert
    ← TradeFailed()
    ← TradeFailed()
    ← TradeFailed()

```

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 712.96ms

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

Failing tests:

Encountered 1 failing test in

```

↳ tests/CrossCurrency/testCrossCurrencyVault.t.sol:TestCrossCurrency_ETH_WSTETH
[FAIL. Reason: TradeFailed(); counterexample:
↳ calldata=0xde6f757a00000000000000000000000000000000000000000000000000000000
↳ 0000000000000000000000000000000000000000000000000000000000000000
↳ args=[0, 0]] test_EnterVault(uint256,uint256) (runs: 0, : 0, ~: 0)

```

Encountered a total of 1 failing tests, 0 tests succeeded

and here is the case where the ETH is transferred to the vault itself, and then `depositFromNotional` is called

```

function enterVaultBypass(
    address account,
    uint256 depositAmount,
    uint256 maturity,
    bytes memory data
) internal virtual returns (uint256 vaultShares) {
    vm.prank(address(NOTIONAL));
    deal(address(vault), depositAmount);
    vaultShares = vault.depositFromNotional(account, depositAmount, maturity,
↳ data);
    totalVaultShares[maturity] += vaultShares;
    totalVaultSharesAllMaturities += vaultShares;
}

```

and here is the trace call, where the call succeeded

```

Running 1 test for
↳ tests/CrossCurrency/testCrossCurrencyVault.t.sol:TestCrossCurrency_ETH_WSTETH

```



```
[PASS] test_EnterVault(uint256,uint256) (runs: 256, : 843050, ~: 982549)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.51s

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

The reason that they didn't find that in the test case is because they were doing the tests in the second way, by transferring the ETH to the Vault with `deal` instead setting a `value` to the call itself, that would be the correct way to test a payable function.

## Czar102

Interesting, thank you @VagnerAndrei26.

What is the loss of funds in this scenario?

## VagnerAndrei26

Interesting, thank you @VagnerAndrei26.

What is the loss of funds in this scenario?

This does not fall under loss of funds here, but more towards the contract rendered useless, since the main functionality, the deposit one, would not work, especially for the pools that have ETH, like the one I provided in the test file.

## jeffywu

@VagnerAndrei26 appreciate the follow up on this issue, I agree it is indeed valid. Good find. I was able to reproduce this below and put a fix in:  
<https://github.com/notional-finance/leveraged-vaults/pull/80>

## Czar102

The reason that they didn't find that in the test case is because they were doing the tests in the second way, by transferring the ETH to the Vault with `deal` instead setting a `value` to the call itself, that would be the correct way to test a payable function.

It seems to me that the same is possible using another way. Also, there is no loss of funds, only loss of functionality. This is a really good find, but per Sherlock rules, I don't think I can make it a medium. Planning to leave it a low severity issue. Appreciate the escalation.

## nevillehuang

Hi @Czar102 based on this comment by @VagnerAndrei26 I think this qualifies for a medium on grounds on a permanent DoS on a desired functionality no?

This does not fall under loss of funds here, but more towards the contract rendered useless, since the main functionality, the deposit one,





would not work, especially for the pools that have ETH, like the one I provided in the test file.

What is the possible other way you are referring to?

### VagnerAndrei26

The reason that they didn't find that in the test case is because they were doing the tests in the second way, by transferring the ETH to the Vault with deal instead setting a value to the call itself, that would be the correct way to test a payable function.

It seems to me that the same is possible using another way. Also, there is no loss of funds, only loss of functionality. This is a really good find, but per Sherlock rules, I don't think I can make it a medium. Planning to leave it a low severity issue. Appreciate the escalation.

Hey @Czar102 , there is not really other ways of doing it. The only way of doing it is passing `msg.value` since it is a function that can only be called from Notional main contract, not by users anytime. They were doing the test in other ways which was not correct, but they fixed the tests also with the fix provided above. Also most of the time, when a contract is rendered useless because of an important functionality, it was considered a medium most of the time on sherlock since it can be based on this rule

2. Breaks core contract functionality, rendering the contract useless (should not be easily replaced without loss of funds) or leading to unknown potential exploits/loss of funds.

Ex: Unable to remove malicious user/collateral from the contract.

since the protocol needs to redeploy most of the contract that were rendered useless. Also it was clearly their intention of working with those pools, and interacting with ether, but because of an important bug it is not working. Personally I can't consider it fair for this to be low/informational, since rendering a contract useless and forcing a redeployments with increased costs to the protocol is not something that should be considered low. Here is one example of a recent contest where, because of an issue inside, the contract was rendering it useless to work with multiple important pools from Curve, which is a similar issue with this one

<https://github.com/sherlock-audit/2023-07-blueberry-judging/issues/105>

### AuditorPraise

The reason that they didn't find that in the test case is because they were doing the tests in the second way, by transferring the ETH to the Vault with deal instead setting a value to the call itself, that would be the correct way to test a payable function.

It seems to me that the same is possible using another way. Also, there is no loss of funds, only loss of functionality. This is a really good find, but per Sherlock rules, I don't think I can make it a medium. Planning to leave it a low severity issue. Appreciate the escalation.



Hello boss @Czar102,

There's no other way to send ETH other than having msg.value > 0. IMO i think this qualifies as a MED due to permanent Dos on a desired functionality.

**Czar102**

Hi, that's right, sorry for my misunderstanding and the confusion because of it. I see how the test didn't fully check the functionality. It's a great find. I think it's a valid medium based on the fact that core functionality is not working at all.

Planning to accept the escalation and make the issue a valid medium.

**Czar102**

Result: Medium Has duplicates

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:

- VagnerAndrei26: accepted

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/80>

**xiaoming9090**

Fixed in PR 80



## Issue M-6: Emergency withdraw might not be enough if the underlying pool is a nested pool

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/52>

### Found by

mstpr-brainbot

### Summary

Emergency withdraw is can be called in various cases. The protocol team states that it can be called when a balancer pool enables "enableRecoveryMode" and maybe it can be called when the pool has an exploit and it needs to be unwinded immediately. However, if the actual pool is a nested pool that has an another pool lp as its underlying and that pool has the exploit or "enableRecoveryMode" toggle on then the emergency withdraw will not be able to exit the position properly since the actual pool that has the emergency issue is the nested pool.

### Vulnerability Detail

Assume that there is a CPS-wstETH-rETH, WETH weighted pool where as this weighted pool has only 2 tokens which is the weth token and the cps-wsteth-reth bpt token. If the emergency withdraw is called the vault will receive those two tokens. However, if the actual problem (maybe exploit or recovery mode toggle on) happens in the underlying pool then the emergency withdraw will not be able to cover the emergency situation.

### Impact

It is likely to have these type of pools and it might make sense to deposit them. However, the emergency withdraw should be properly seated for these type pools to maintain the emergency functionality. Since the protocol team states that any pool that has a notional finance assets is listed is eligible for single sided strategy, and in such pools the emergency withdraw can be broken I will label this as medium.

### Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/7aadd254da5f645a7e1b718e7f9128f845e10f02/leveraged-vaults/contracts/vaults/common/SingleSidedLPVaultBase.sol#L480-L496>



## Tool used

Manual Review

## Recommendation

If the pool is nested make the emergency exit also exits the underlying pool if the emergency role wants to. That way the emergency functionality can work properly for nested pools.

## Discussion

**jeffywu**

This is a fair point.

**jeffywu**

Fix here is to add a check to ensure that nested pools are not included:  
<https://github.com/notional-finance/leveraged-vaults/pull/72>

**MLON33**

<https://github.com/notional-finance/leveraged-vaults/pull/72>

**xiaoming9090**

Fixed in [PR 72](#)



## Issue M-7: ETH can be sold during reinvestment

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/74>

### Found by

lemonmon, xiaoming90

### Summary

The existing control to prevent ETH from being sold during reinvestment can be bypassed, allowing the bots to accidentally or maliciously sell off the non-reward assets of the vault.

### Vulnerability Detail

Multiple instances of this issue were found:

#### Instance 1 - Curve's Implementation

The `_isInvalidRewardToken` function attempts to prevent the callers from selling away ETH during reinvestment.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/curve/ConvexStakingMixin.sol#L60>

```
File: ConvexStakingMixin.sol
60:     function _isInvalidRewardToken(address token) internal override view
    ↪ returns (bool) {
61:         return (
62:             token == TOKEN_1 ||
63:             token == TOKEN_2 ||
64:             token == address(CURVE_POOL_TOKEN) ||
65:             token == address(CONVEX_REWARD_POOL) ||
66:             token == address(CONVEX_BOOSTER) ||
67:             token == Deployments.ALT_ETH_ADDRESS
68:         );
69:     }
```

However, the code at Line 67 above will not achieve the intended outcome as `Deployments.ALT_ETH_ADDRESS` is not a valid token address in the first place.

```
address internal constant ALT_ETH_ADDRESS =
    ↪ 0xEeeeeEeeeEeEeeEeEeEEEEEEEEEEEEEEEEEE;
```



When the caller is executing a trade with ETH, the address for ETH used is either `Deployments.WETH` OR `Deployments.ETH_ADDRESS` (`address(0)`) as shown in the `TradingUtils`'s source code, not the `Deployments.ALT_ETH_ADDRESS`.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/trading/TradingUtils.sol#L128>

```
File: TradingUtils.sol
128:     function _executeTrade(
129:         address target,
130:         uint256 msgValue,
131:         bytes memory params,
132:         address spender,
133:         Trade memory trade
134:     ) private {
135:         uint256 preTradeBalance;
136:
137:         if (trade.buyToken == address(Deployments.WETH)) {
138:             preTradeBalance = address(this).balance;
139:         } else if (trade.buyToken == Deployments.ETH_ADDRESS ||
↳ _needsToUnwrapExcessWETH(trade, spender)) {
140:             preTradeBalance =
↳ IERC20(address(Deployments.WETH)).balanceOf(address(this));
141:         }
```

As a result, the caller (bot) of the reinvestment function could still sell off the ETH from the vault, bypassing the requirement.

## Instance 2 - Balancer's Implementation

When the caller is executing a trade with ETH, the address for ETH used is either `Deployments.WETH` OR `Deployments.ETH_ADDRESS` (`address(0)`), as mentioned earlier. However, the `AuraStakingMixin._isInvalidRewardToken` function only blocks `Deployments.WETH` but not `Deployments.ETH`, thus allowing the caller (bot) of the reinvestment function, could still sell off the ETH from the vault, bypassing the requirement.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/mixins/AuraStakingMixin.sol#L38>

```
File: AuraStakingMixin.sol
38:     function _isInvalidRewardToken(address token) internal override view
↳ returns (bool) {
39:         return (
40:             token == TOKEN_1 ||
41:             token == TOKEN_2 ||
42:             token == TOKEN_3 ||
```



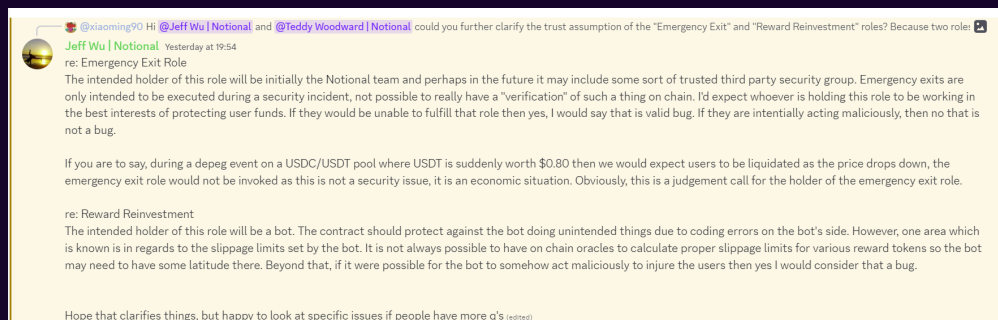
```

43:         token == TOKEN_4 ||
44:         token == TOKEN_5 ||
45:         token == address(AURA_BOOSTER) ||
46:         token == address(AURA_REWARD_POOL) ||
47:         token == address(Deployments.WETH)
48:     );
49: }

```

Per the sponsor's clarification below, the contracts should protect against the bot doing unintended things (including acting maliciously) due to coding errors, which is one of the main reasons for having the `_isInvalidRewardToken` function. Thus, this issue is a valid bug in the context of this audit contest.

<https://discord.com/channels/812037309376495636/1175450365395751023/1175781082336067655>



## Impact

The existing control to prevent ETH from being sold during reinvestment can be bypassed, allowing the bots to accidentally or maliciously sell off the non-reward assets of the vault.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/curve/ConvexStakingMixin.sol#L60>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/mixins/AuraStakingMixin.sol#L38>

## Tool used

Manual Review

## Recommendation

To ensure that ETH cannot be sold off during reinvestment, consider the following changes:

### Curve

```
File: ConvexStakingMixin.sol
function _isInvalidRewardToken(address token) internal override view returns
↳ (bool) {
    return (
        token == TOKEN_1 ||
        token == TOKEN_2 ||
        token == address(CURVE_POOL_TOKEN) ||
        token == address(CONVEX_REWARD_POOL) ||
        token == address(CONVEX_BOOSTER) ||
+       token == Deployments.ETH ||
+       token == Deployments.WETH ||
        token == Deployments.ALT_ETH_ADDRESS
    );
}
```

### Balancer

```
File: AuraStakingMixin.sol
function _isInvalidRewardToken(address token) internal override view returns
↳ (bool) {
    return (
        token == TOKEN_1 ||
        token == TOKEN_2 ||
        token == TOKEN_3 ||
        token == TOKEN_4 ||
        token == TOKEN_5 ||
        token == address(AURA_BOOSTER) ||
        token == address(AURA_REWARD_POOL) ||
+       token == address(Deployments.ETH)
        token == address(Deployments.WETH)
    );
}
```

## Discussion

jeffyu

Valid report.

jeffyu





<https://github.com/notional-finance/leveraged-vaults/pull/67>

**MLON33**

<https://github.com/notional-finance/leveraged-vaults/pull/67>

**xiaoming9090**

Fixed in [PR 67](#)



## Issue M-8: BPT LP Token could be sold off during re-investment

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/76>

### Found by

xiaoming90

### Summary

BPT LP Token could be sold off during the re-investment process. BPT LP Tokens must not be sold to external DEXs under any circumstance because:

- They are used to redeem the underlying assets from the pool when someone exits the vault
- The BPTs represent the total value of the vault

### Vulnerability Detail

Within the `ConvexStakingMixin._isInvalidRewardToken` function, the implementation ensures that the LP Token (`CURVE_POOL_TOKEN`) is not intentionally or accidentally sold during the reinvestment process.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/curve/ConvexStakingMixin.sol#L64>

```
File: ConvexStakingMixin.sol
60:     function _isInvalidRewardToken(address token) internal override view
    ↪ returns (bool) {
61:         return (
62:             token == TOKEN_1 ||
63:             token == TOKEN_2 ||
64:             token == address(CURVE_POOL_TOKEN) ||
65:             token == address(CONVEX_REWARD_POOL) ||
66:             token == address(CONVEX_BOOSTER) ||
67:             token == Deployments.ALT_ETH_ADDRESS
68:         );
69:     }
```

However, the same control was not implemented for the Balancer/Aura code. As a result, it is possible for LP Token (BPT) to be sold during reinvestment. Note that for non-composable Balancer pools, the pool tokens does not consists of the BPT token. Thus, it needs to be explicitly defined within the `_isInvalidRewardToken` function.

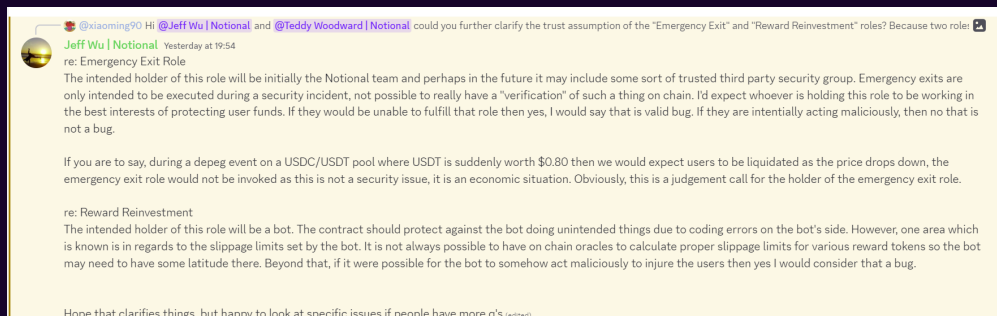


<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/mixins/AuraStakingMixin.sol#L38>

```
File: AuraStakingMixin.sol
38:     function _isInvalidRewardToken(address token) internal override view
    ↪ returns (bool) {
39:         return (
40:             token == TOKEN_1 ||
41:             token == TOKEN_2 ||
42:             token == TOKEN_3 ||
43:             token == TOKEN_4 ||
44:             token == TOKEN_5 ||
45:             token == address(AURA_BOOSTER) ||
46:             token == address(AURA_REWARD_POOL) ||
47:             token == address(Deployments.WETH)
48:         );
49:     }
```

Per the sponsor's clarification below, the contracts should protect against the bot doing unintended things (including acting maliciously) due to coding errors, which is one of the main reasons for having the `_isInvalidRewardToken` function. Thus, this issue is a valid bug in the context of this audit contest.

<https://discord.com/channels/812037309376495636/1175450365395751023/1175781082336067655>



The screenshot shows a Discord message from Jeff Wu | Notional. The message is a reply to a question about the trust assumption of the "Emergency Exit" and "Reward Reinvestment" roles. It contains two sections: "re: Emergency Exit Role" and "re: Reward Reinvestment".

**re: Emergency Exit Role**

The intended holder of this role will be initially the Notional team and perhaps in the future it may include some sort of trusted third party security group. Emergency exits are only intended to be executed during a security incident, not possible to really have a "verification" of such a thing on chain. I'd expect whoever is holding this role to be working in the best interests of protecting user funds. If they would be unable to fulfill that role then yes, I would say that is valid bug. If they are intentionally acting maliciously, then no that is not a bug.

If you are to say, during a depeg event on a USDC/USDT pool where USDT is suddenly worth \$0.80 then we would expect users to be liquidated as the price drops down, the emergency exit role would not be invoked as this is not a security issue, it is an economic situation. Obviously, this is a judgement call for the holder of the emergency exit role.

**re: Reward Reinvestment**

The intended holder of this role will be a bot. The contract should protect against the bot doing unintended things due to coding errors on the bot's side. However, one area which is known is in regards to the slippage limits set by the bot. It is not always possible to have on chain oracles to calculate proper slippage limits for various reward tokens so the bot may need to have some latitude there. Beyond that, if it were possible for the bot to somehow act maliciously to injure the users then yes I would consider that a bug.

Hope that clarifies things, but happy to look at specific issues if people have more q's (edited)

## Impact

LP tokens (BPT) might be accidentally or maliciously sold off by the bots during the re-investment process. BPT LP Tokens must not be sold to external DEXs under any circumstance because:

- They are used to redeem the underlying assets from the pool when someone exits the vault
- The BPTs represent the total value of the vault



## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/balancer/mixins/AuraStakingMixin.sol#L38>

## Tool used

Manual Review

## Recommendation

Ensure that the LP tokens cannot be sold off during re-investment.

```
function _isInvalidRewardToken(address token) internal override view returns
↳ (bool) {
    return (
        token == TOKEN_1 ||
        token == TOKEN_2 ||
        token == TOKEN_3 ||
        token == TOKEN_4 ||
        token == TOKEN_5 ||
+       token == BALANCER_POOL_TOKEN ||
        token == address(AURA_BOOSTER) ||
        token == address(AURA_REWARD_POOL) ||
        token == address(Deployments.WETH)
    );
}
```

## Discussion

**jeffyu**

Valid issue.

**jeffyu**

<https://github.com/notional-finance/leveraged-vaults/pull/66>

**MLON33**

Fix: <https://github.com/notional-finance/leveraged-vaults/pull/66>

**xiaoming9090**

Fixed in [PR 66](#)



## Issue M-9: Leverage Vault on sidechains that support Curve V2 pools is broken

Source: <https://github.com/sherlock-audit/2023-10-notional-judging/issues/88>

### Found by

xiaoming90

### Summary

No users will be able to deposit to the Leverage Vault on Arbitrum and Optimism that supports Curve V2 pools, leading to the core contract functionality of a vault being broken and a loss of revenue for the protocol.

### Vulnerability Detail

Following are examples of some Curve V2 pools in Arbitrum:

- fETH/ETH/xETH (<https://curve.fi/#/arbitrum/pools/factory-tricrypto-2/deposit>)
- tricrypto (<https://curve.fi/#/arbitrum/pools/tricrypto/deposit>)
- eursusd (<https://curve.fi/#/arbitrum/pools/eursusd/deposit>)

The code from Line 64 to Line 71 is only executed if the contract resides on Ethereum. As a result, for Arbitrum and Optimism sidechains, the `IS_CURVE_V2` variable is always false.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/curve/Curve2TokenPoolMixin.sol#L73>

```
File: Curve2TokenPoolMixin.sol
56:     constructor(
57:         NotionalProxy notional_,
58:         DeploymentParams memory params
59:     ) SingleSidedLPVaultBase(notional_, params.tradingModule) {
60:         CURVE_POOL = params.pool;
61:
62:         bool isCurveV2 = false;
63:         if (Deployments.CHAIN_ID == Constants.CHAIN_ID_MAINNET) {
64:             address[10] memory handlers =
65:                 Deployments.CURVE_META_REGISTRY.get_registry_handlers_from_p
↳ ool(address(CURVE_POOL));
66:
67:             require(
```



```

68:             handlers[0] == Deployments.CURVE_V1_HANDLER ||
69:             handlers[0] == Deployments.CURVE_V2_HANDLER
70:         ); // @dev unknown Curve version
71:         isCurveV2 = (handlers[0] == Deployments.CURVE_V2_HANDLER);
72:     }
73:     IS_CURVE_V2 = isCurveV2;

```

As a result, code within the `_joinPoolAndStake` function will always call the Curve V1's `add_liquidity` function that does not define the `use_eth` parameter.

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol#L51>

```

File: Curve2TokenConvexVault.sol
26:     function _joinPoolAndStake(
..SNIP..
45:         // Slightly different method signatures in v1 and v2
46:         if (IS_CURVE_V2) {
47:             lpTokens = ICurve2TokenPoolV2(CURVE_POOL).add_liquidity{value:
↳ msgValue}(
48:                 amounts, minPoolClaim, 0 < msgValue // use_eth = true if
↳ msgValue > 0
49:             );
50:         } else {
51:             lpTokens = ICurve2TokenPoolV1(CURVE_POOL).add_liquidity{value:
↳ msgValue}(
52:                 amounts, minPoolClaim
53:             );
54:         }

```

If the `use_eth` parameter is not defined, it will default to `False`. As a result, the Curve pool expects the caller to transfer over the WETH to the pool and the pool will call `WETH.withdraw` to unwrap the WETH to Native ETH as shown in the code below.

However, Notional's leverage vault only works with Native ETH, and if one of the pool tokens is WETH, it will explicitly convert the address to either the `Deployments.ALT_ETH_ADDRESS (0xEeeee)` or `Deployments.ETH_ADDRESS (address(0))` during deployment and initialization.

The implementation of the above `_joinPoolAndStake` function will forward Native ETH to the Curve Pool, while the pool expects the vault to transfer in WETH. As a result, a revert will occur since the pool did not receive the WETH it required during the unwrap process.

<https://arbiscan.io/address/0xf7fed8ae0c5b78c19aadd68b700696933b0cefd9#code#L509> (Taken from Curve V2 fETH/ETH/xETH pool)



```

def add_liquidity(
    amounts: uint256[N_COINS],
    min_mint_amount: uint256,
    use_eth: bool = False,
    receiver: address = msg.sender
) -> uint256:
    """
    @notice Adds liquidity into the pool.
    @param amounts Amounts of each coin to add.
    @param min_mint_amount Minimum amount of LP to mint.
    @param use_eth True if native token is being added to the pool.
    @param receiver Address to send the LP tokens to. Default is msg.sender
    @return uint256 Amount of LP tokens received by the `receiver`
    """
    ..SNIP..
    # ----- Get prices, balances -----
    ..SNIP..
    # ----- Update balances and calculate xp.
    ..SNIP...
    # ----- transferFrom token into the pool -----

    for i in range(N_COINS):

        if amounts[i] > 0:

            if coins[i] == WETH20:

                self._transfer_in(
                    coins[i],
                    amounts[i],
                    0, # <-----
                    msg.value, # | No callbacks
                    empty(address), # <-----| for
                    empty(bytes32), # <-----| add_liquidity.
                    msg.sender, # |
                    empty(address), # <-----
                    use_eth
                )

```

```

def _transfer_in(
    ..SNIP..
    use_eth: bool
):
    ..SNIP..
    @params use_eth True if the transfer is ETH, False otherwise.

```



```

    """

    if use_eth and _coin == WETH20:
        assert mvalue == dx # dev: incorrect eth amount
    else:
        ..SNIP..
        if _coin == WETH20:
            WETH(WETH20).withdraw(dx) # <----- if WETH was transferred in
            # previous step and `not use_eth`, withdraw WETH to ETH.

```

## Impact

No users will be able to deposit to the Leverage Vault on Arbitrum and Optimism that supports Curve V2 pools. The deposit function is a core function of any vault. Thus, this issue breaks the core contract functionality of a vault.

In addition, if the affected vaults cannot be used, it leads to a loss of revenue for the protocol.

## Code Snippet

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/curve/Curve2TokenPoolMixin.sol#L73>

<https://github.com/sherlock-audit/2023-10-notional/blob/main/leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol#L51>

## Tool used

Manual Review

## Recommendation

Ensure the `IS_CURVE_V2` variable is initialized on the Arbitrum and Optimism side chains according to the Curve Pool's version.

If there is a limitation on the existing approach to determining a pool is V1 or V2 on Arbitrum and Optimism, an alternative approach might be to use the presence of a `gamma()` function as an indicator of pool type

## Discussion

jeffyu

It should be noted that none of the pools are available on Convex either:  
<https://www.convexfinance.com/stake>





Furthermore, the Convex vault is only explicitly written for 2 token vaults, which the ones auditor listed are not. So therefore they could not be listed as structured Arbitrum in any case. I would range this as a medium severity, if anything.

