SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Notional
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90
Dates Audited:	March 1 - March 8, 2023
Prepared on:	April 1, 2023

Introduction

Earn fixed income on your crypto or borrow at fixed rates for up to one year with Notional - DeFi's top fixed rate protocol.

Scope

leveraged-vaults @ ec790f931988904f99da5c3514e8e1c74bad050b

- leveraged-vaults/contracts/trading/adapters/CurveV2Adapter.sol
- leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol
- leveraged-vaults/contracts/vaults/curve/CurveVaultTypes.sol
- leveraged-vaults/contracts/vaults/curve/external/Curve2TokenConvexHelper.sol
- leveraged-vaults/contracts/vaults/curve/internal/CurveConstants.sol
- leveraged-vaults/contracts/vaults/curve/internal/CurveVaultStorage.sol
- leveraged-vaults/contracts/vaults/curve/internal/pool/Curve2TokenPoolUtils.sol
- leveraged-vaults/contracts/vaults/curve/mixins/ConvexStakingMixin.sol
- leveraged-vaults/contracts/vaults/curve/mixins/Curve2TokenPoolMixin.sol
- leveraged-vaults/contracts/vaults/curve/mixins/Curve2TokenVaultMixin.sol
- leveraged-vaults/contracts/vaults/curve/mixins/CurvePoolMixin.sol
- leveraged-vaults/contracts/trading/adapters/CurveAdapter.sol

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
5	6

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

xiaoming90 usmannk

Issue H-1: Liquidations are impossible for some Curve pools

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/21

Found by

usmannk

Summary

Some curve pools have implementations such that Notional liquidations always revert.

Vulnerability Detail

Liquidations are done, directly or indirectly, via the `deleverageAccount` function. This function calls `_checkReentrancyContext` to protect against read-only reentrancy.

The Curve vault's `_checkReentrancyContext` function uses the Curve `remove_liquidity` function to check the reentrancy context. However, for certain Curve pools like the CRV/ETH pool (0x8301ae4fc9c624d1d396cbdaa1ed877821d7c511, https://curve.fi/#/ethereum/pools/crveth/) calling `remove_liquidity(0, [0,0])` always reverts due to an underflow.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/mixins/Curve2TokenVaultMixin.sol#L13-L16

Impact

Liquidations are not possible, users can go into bad debt and there is no way to recover the lost funds.

Code Snippet

Tool used

Manual Review

Recommendation

Use the `claim_admin_fees` function to check Curve's reentrancy state instead of `remove_liquidity`.

Discussion

jeffywu

Valid, appears that removing 1 token will be sufficient to pass the underflow check. We need to make a note of this and ensure that we either pass in a parameter of 1 or 0 based on the target pool.

Issue H-2: Risk of reward tokens being sold by malicious users under certain conditions

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/13

Found by

xiaoming90

Summary

Due to the lack of validation of the selling token within the Curve adaptors, there is a risk that the reward tokens or Convex deposit tokens of the vault being sold by malicious users under certain conditions (e.g. if reward tokens equal to primary/secondary tokens OR a new exploit is found in other parts of the code).

Vulnerability Detail

For a `EXACT_IN_SINGLE` trade within the Curve adaptors, the `from` and `to` addresses of the `exchange` function are explicitly set to `trade.sellToken` and `trade.buyToken` respectively. Thus, the swap is restricted to only `trade.sellToken` and `trade.buyToken`, which points to either the primary or secondary token of the pool. This prevents other tokens that reside in the vault from being swapped out.

However, this measure was not applied to the `EXACT_IN_BATCH` trade as it ignores the `trade.sellToken` and `trade.buyToken` , and allow the caller to define arbitrary `data.route` where the first route (`_route[0]`) and last route (`_route[last_index]`) could be any token.

The vault will hold the reward tokens (CRV, CVX, LDO) when the vault administrator claims the rewards or a malicious user claims the rewards on behalf of the vault by calling Convex's getReward function.

Assume that attacker is faster than the admin calling the reinvest function. There is a possibility that an attacker executes a `EXACT_IN_BATCH` trade and specifies the `_route[0]` as one of the reward tokens residing on the vault and swaps away the reward tokens during depositing (`_tradePrimaryForSecondary`) or redemption (`_sellSecondaryBalance`). In addition, an attacker could also sell away the Convex deposit tokens if a new exploit is found.

In addition, the vault also holds Convex deposit tokens, which represent assets held by the vault.

This issue affects the in-scope `CurveV2Adapter` and `CurveAdapter` since they do not validate the `data.route` provided by the users.

CurveV2Adapter https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/trading/adapters/CurveV2Adapter.sol#L37

```
File: CurveV2Adapter.sol
37:     function getExecutionData(address from, Trade calldata trade)
38:         internal view returns (
39:             address spender,
40:             address target,
41:             uint256 msgValue,
42:             bytes memory executionCallData
43:         )
44:     {
45:         if (trade.tradeType == TradeType.EXACT_IN_SINGLE) {
46:             CurveV2SingleData memory data = abi.decode(trade.exchangeData,
 ↪  (CurveV2SingleData));
47:             executionCallData = abi.encodeWithSelector(
48:                 ICurveRouterV2.exchange.selector,
49:                 data.pool,
50:                 _getTokenAddress(trade.sellToken),
51:                 _getTokenAddress(trade.buyToken),
52:                 trade.amount,
53:                 trade.limit,
54:                 address(this)
55:             );
56:         } else if (trade.tradeType == TradeType.EXACT_IN_BATCH) {
57:             CurveV2BatchData memory data = abi.decode(trade.exchangeData,
 ↪  (CurveV2BatchData));
58:             // Array of pools for swaps via zap contracts. This parameter is
 ↪  only needed for
59:             // Polygon meta-factories underlying swaps.
60:             address[4] memory pools;
61:             executionCallData = abi.encodeWithSelector(
62:                 ICurveRouterV2.exchange_multiple.selector,
63:                 data.route,
64:                 data.swapParams,
65:                 trade.amount,
66:                 trade.limit,
67:                 pools,
68:                 address(this)
69:             );
```

CurveAdapter https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/trading/adapters/CurveAdapter.sol#L66

```
File: CurveAdapter.sol
22:     function _exactInBatch(Trade memory trade) internal view returns (bytes
 ↪  memory executionCallData) {
```

```
23:         CurveBatchData memory data = abi.decode(trade.exchangeData,
↪  (CurveBatchData));
24:
25:         return abi.encodeWithSelector(
26:             ICurveRouter.exchange.selector,
27:             trade.amount,
28:             data.route,
29:             data.indices,
30:             trade.limit
31:         );
32:     }
```

Following are some examples of where this vulnerability could potentially be exploited. Assume a vault that supports the CurveV2's ETH/stETH pool.

1) Perform the smallest possible redemption to trigger the `_sellSecondaryBalance` function. Configure the `RedeemParams` to swap the reward token (CRV, CVX, or LDO) or Convex Deposit token for the primary token (ETH). This will cause the `finalPrimaryBalance` to increase by the number of incoming primary tokens (ETH), thus inflating the number of primary tokens redeemed.
2) Perform the smallest possible deposit to trigger the `_tradePrimaryForSecondary`. Configure `DepositTradeParams` to swap the reward token (CRV, CVX, or LDO) or Convex Deposit token for the secondary tokens (stETH). This will cause the `secondaryAmount` to increase by the number of incoming secondary tokens (stETH), thus inflating the number of secondary tokens available for the deposit.

Upon further investigation, it was observed that the vault would only approve the exchange to pull the `trade.sellToken`, which points to either the primary token (ETH) or secondary token (stETH). Thus, the reward tokens (CRV, CVX, or LDO) or Convex deposit tokens cannot be sent to the exchanges. Thus, the vault will not be affected if none of the reward tokens/Convex Deposit tokens equals the primary or secondary token.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/trading/TradingUtils.sol#L118

```
File: TradingUtils.sol
115:     /// @notice Approve exchange to pull from this contract
116:     /// @dev approve up to trade.amount for EXACT_IN trades and up to
↪  trade.limit
117:     /// for EXACT_OUT trades
118:     function _approve(Trade memory trade, address spender) private {
119:         uint256 allowance = _isExactIn(trade) ? trade.amount : trade.limit;
120:         address sellToken = trade.sellToken;
121:         // approve WETH instead of ETH for ETH trades if
```

```
122:         // spender != address(0) (checked by the caller)
123:         if (sellToken == Constants.ETH_ADDRESS) {
124:             sellToken = address(Deployments.WETH);
125:         }
126:         IERC20(sellToken).checkApprove(spender, allowance);
127:     }
```

However, there might be some Curve Pools or Convex's reward contracts whose reward tokens are similar to the primary or secondary tokens of the vault. If the vault supports those pools, the vault will be vulnerable. In addition, the reward tokens of a Curve pool or Convex's reward contracts are not immutable. It is possible for the governance to add a new reward token that might be the same as the primary or secondary token.

Impact

There is a risk that the reward tokens or Convex deposit tokens of the vault are sold by malicious users under certain conditions (e.g. if reward tokens are equal to primary/secondary tokens OR a new exploit is found in other parts of the code), thus potentially draining assets from the vault.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/trading/adapters/CurveV2Adapter.sol#L37

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/trading/adapters/CurveAdapter.sol#L66

Tool used

Manual Review

Recommendation

It is recommended to implement additional checks when performing a `EXACT_IN_BATCH` trade with the `CurveV2Adapter` or `CurveAdapter` adaptor. The first item in the route must be the `trade.sellToken`, and the last item in the route must be the `trade.buyToken`. This will restrict the `trade.sellToken` to the primary or secondary token, and prevent reward and Convex Deposit tokens from being sold (Assuming primary/secondary token != reward tokens).

```
route[0] == trade.sellToken
route[last index] == trade.buyToken
```

The vault holds many Convex Deposit tokens (e.g. cvxsteCRV). A risk analysis of the vault shows that the worst thing that could happen is that all the Convex Deposit tokens are swapped away if a new exploit is found, which would drain the entire vault. For defense-in-depth, it is recommended to check that the selling token is not a Convex Deposit token under any circumstance when using the trade adaptor.

The trade adaptors are one of the attack vectors that the attacker could potentially use to move tokens out of the vault if any exploit is found. Thus, they should be locked down or restricted where possible.

Alternatively, consider removing the `EXACT_IN_BATCH` trade function from the affected adaptors to reduce the attack surface if the security risk of this feature outweighs the benefit of the batch function.

Discussion

jeffywu

Valid, agree that require checks needed here. @weitianjie2000, also review that the other adapters have similar require checks for batch trades.

Issue H-3: Ineffective slippage mechanism when redeeming proportionally

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/12

Found by

xiaoming90

Summary

A trade will continue to be executed regardless of how bad the slippage is since the minimum amount returned by the `TwoTokenPoolUtils._getMinExitAmounts` function does not work effectively. Thus, a trade might incur significant slippage, resulting in the vault receiving fewer tokens in return, leading to losses for the vault shareholders.

Vulnerability Detail

The `params.minPrimary` and `params.minSecondary` are calculated automatically based on the share of the Curve pool with a small discount within the `Curve2TokenConvexHelper._executeSettlement` function (Refer to Line 124 below)

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/external/Curve2TokenConvexHelper.sol#L112

```
File: Curve2TokenConvexHelper.sol
112:     function _executeSettlement(
113:         StrategyContext calldata strategyContext,
114:         Curve2TokenPoolContext calldata poolContext,
115:         uint256 maturity,
116:         uint256 poolClaimToSettle,
117:         uint256 redeemStrategyTokenAmount,
118:         RedeemParams memory params
119:     ) private {
120:         (uint256 spotPrice, uint256 oraclePrice) =
 ↪  poolContext._getSpotPriceAndOraclePrice(strategyContext);
121:
122:         /// @notice params.minPrimary and params.minSecondary are not
 ↪  required to be passed in by the caller
123:         /// for this strategy vault
124:         (params.minPrimary, params.minSecondary) =
 ↪  poolContext.basePool._getMinExitAmounts({
125:             strategyContext: strategyContext,
126:             oraclePrice: oraclePrice,
127:             spotPrice: spotPrice,
```

```
128:             poolClaim: poolClaimToSettle
129:         });
```

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/pool/TwoTokenPoolUtils.sol#L48

```
File: TwoTokenPoolUtils.sol
46:     /// @notice calculates the expected primary and secondary amounts based
 ↪  on
47:     /// the given spot price and oracle price
48:     function _getMinExitAmounts(
49:         TwoTokenPoolContext calldata poolContext,
50:         StrategyContext calldata strategyContext,
51:         uint256 spotPrice,
52:         uint256 oraclePrice,
53:         uint256 poolClaim
54:     ) internal view returns (uint256 minPrimary, uint256 minSecondary) {
55:         strategyContext._checkPriceLimit(oraclePrice, spotPrice);
56:
57:         // min amounts are calculated based on the share of the Balancer
 ↪  pool with a small discount applied
58:         uint256 totalPoolSupply = poolContext.poolToken.totalSupply();
        minPrimary = (poolContext.primaryBalance * poolClaim *
            strategyContext.vaultSettings.poolSlippageLimitPercent) / //
 ↪  @audit-info poolSlippageLimitPercent = 9975, # 0.25%
            (totalPoolSupply * uint256(VaultConstants.VAULT_PERCENT_BASIS)); //
 ↪  @audit-info VAULT_PERCENT_BASIS = 1e4 = 10000
62:         minSecondary = (poolContext.secondaryBalance * poolClaim *
63:             strategyContext.vaultSettings.poolSlippageLimitPercent) /
64:             (totalPoolSupply * uint256(VaultConstants.VAULT_PERCENT_BASIS));
65:     }
```

When LP tokens are redeemed proportionally via the Curve Pool's remove_liquidity function, the tokens received are based on the share of the Curve pool as the source code.

```
@external
@nonreentrant('lock')
def remove_liquidity(
    _amount: uint256,
    _min_amounts: uint256[N_COINS],
) -> uint256[N_COINS]:
    """
    @notice Withdraw coins from the pool
    @dev Withdrawal amounts are based on current deposit ratios
    @param _amount Quantity of LP tokens to burn in the withdrawal
    @param _min_amounts Minimum amounts of underlying coins to receive
```

```
    @return List of amounts of coins that were withdrawn
    """
    amounts: uint256[N_COINS] = self._balances()
    lp_token: address = self.lp_token
    total_supply: uint256 = ERC20(lp_token).totalSupply()
    CurveToken(lp_token).burnFrom(msg.sender, _amount)  # dev: insufficient funds

    for i in range(N_COINS):
        value: uint256 = amounts[i] * _amount / total_supply
        assert value >= _min_amounts[i], "Withdrawal resulted in fewer coins
        ↪  than expected"

        amounts[i] = value
        if i == 0:
            raw_call(msg.sender, b"", value=value)
        else:
            assert ERC20(self.coins[1]).transfer(msg.sender, value)

    log RemoveLiquidity(msg.sender, amounts, empty(uint256[N_COINS]),
    ↪  total_supply - _amount)

    return amounts
```

Assume a Curve Pool with the following state:

- Consists of 200 US Dollars worth of tokens (100 DAI and 100 USDC). DAI is the primary token
- DAI <> USDC price is 1:1
- Total Supply = 100 LP Pool Tokens

Assume that 50 LP Pool Tokens will be claimed during vault settlement.

`TwoTokenPoolUtils._getMinExitAmounts` function will return `49.875 DAI` as `params.minPrimary` and `49.875 USDC` as `params.minSecondary` based on the following calculation

```
minPrimary = (poolContext.primaryBalance * poolClaim *
↪  strategyContext.vaultSettings.poolSlippageLimitPercent /  (totalPoolSupply *
↪  uint256(VaultConstants.VAULT_PERCENT_BASIS)
minPrimary = (100 DAI * 50 LP_TOKEN * 99.75% /  (100 LP_TOKEN * 100%)

Rewrite for clarity (ignoring rounding error):
minPrimary = 100 DAI * (50 LP_TOKEN/100 LP_TOKEN) * (99.75%/100%) = 49.875 DAI

minSecondary = same calculation = 49.875 USDC
```

Curve Pool's remove_liquidity function will return `50 DAI` and `50 USDC` if 50 LP Pool

Tokens are redeemed.

Note that `TwoTokenPoolUtils._getMinExitAmounts` function performs the calculation based on the spot balance of the pool similar to the approach of the Curve Pool's remove_liquidity function. However, the `TwoTokenPoolUtils._getMinExitAmounts` function applied a discount to the returned result, while the Curve Pool's remove_liquidity function did not.

As such, the number of tokens returned by Curve Pool's remove_liquidity function will always be larger than the number of tokens returned by the `TwoTokenPoolUtils._getMinExitAmounts` function regardless of the on-chain economic condition or the pool state (e.g. imbalance). Thus, the minimum amounts (`minAmounts`) pass into the Curve Pool's `remove_liquidity` function will never be triggered under any circumstance.

```
a = Curve Pool's remove_liquidity => x DAI
b = TwoTokenPoolUtils._getMinExitAmounts => (x DAI - 0.25% discount)
a > b => true (for all instances)
```

Thus, the `TwoTokenPoolUtils._getMinExitAmounts` function is not effective in determining the slippage when redeeming proportionally.

Impact

A trade will always be executed even if it returns fewer than expected assets since the minimum amount returned by the `TwoTokenPoolUtils._getMinExitAmounts` function does not work effectively. Thus, a trade might incur unexpected slippage, resulting in the vault receiving fewer tokens in return, leading to losses for the vault shareholders.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/external/Curve2TokenConvexHelper.sol#L112

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/pool/TwoTokenPoolUtils.sol#L48

Tool used

Manual Review

Recommendation

When redeeming proportional, the`TwoTokenPoolUtils._getMinExitAmounts` function can be removed. Instead, give the caller the flexibility to define the

slippage/minimum amount (`params.minPrimary` and `params.minSecondary`). To prevent the caller from setting a slippage that is too large, consider restricting the slippage to an acceptable range.

The proper way of computing the minimum amount of tokens to receive from a proportional trade (`remove_liquidity`) is to call the Curve's Pool calc_token_amount function off-chain and reduce the values returned by the allowed slippage amount.

Note that `calc_token_amount` cannot be used solely on-chain for computing the minimum amount because the result can be manipulated because it uses spot balances for computation.

Sidenote: Removing `TwoTokenPoolUtils._getMinExitAmounts` function also removes the built-in spot price and oracle price validation. Thus, the caller must remember to define the slippage. Otherwise, the vault settlement will risk being sandwiched. Alternatively, shift the `strategyContext._checkPriceLimit(oraclePrice, spotPrice)` code outside the `TwoTokenPoolUtils._getMinExitAmounts` function.

Discussion

jeffywu

Valid, min amounts should be specified by the caller to eliminate the risk of sandwich attacks.

Issue H-4: Slippage/Minimum amount does not work during single-side redemption

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/10

Found by

xiaoming90

Summary

The slippage or minimum amount of tokens to be received is set to a value much smaller than expected due to the use of `TwoTokenPoolUtils._getMinExitAmounts` function to automatically compute the slippage or minimum amount on behalf of the callers during a single-sided redemption. As a result, the vault will continue to redeem the pool tokens even if the trade incurs significant slippage, resulting in the vault receiving fewer tokens in return, leading to losses for the vault shareholders.

Vulnerability Detail

The `Curve2TokenConvexHelper._executeSettlement` function is called by the following functions:

- `Curve2TokenConvexHelper.settleVault`
  - `Curve2TokenConvexHelper.settleVault` function is called within the `Curve2TokenConvexVault.settleVaultNormal` and `Curve2TokenConvexVault.settleVaultPostMaturity` functions
- `Curve2TokenConvexHelper.settleVaultEmergency`
  - `Curve2TokenConvexHelper.settleVaultEmergency` is called by `Curve2TokenConvexVault.settleVaultEmergency`

In summary, the `Curve2TokenConvexHelper._executeSettlement` function is called during vault settlement.

An important point to note here is that within the `Curve2TokenConvexHelper._executeSettlement` function, the `params.minPrimary` and `params.minSecondary` are automatically computed and overwritten by the `TwoTokenPoolUtils._getMinExitAmounts` function (Refer to Line 124 below). Therefore, if the caller attempts to define the `params.minPrimary` and `params.minSecondary`, they will be discarded and overwritten. The `params.minPrimary` and `params.minSecondary` is for slippage control when redeeming the Curve's LP tokens.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/external/Curve2TokenConvexHelper.sol#L112

```
File: Curve2TokenConvexHelper.sol
112:     function _executeSettlement(
113:         StrategyContext calldata strategyContext,
114:         Curve2TokenPoolContext calldata poolContext,
115:         uint256 maturity,
116:         uint256 poolClaimToSettle,
117:         uint256 redeemStrategyTokenAmount,
118:         RedeemParams memory params
119:     ) private {
120:         (uint256 spotPrice, uint256 oraclePrice) =
 ↪  poolContext._getSpotPriceAndOraclePrice(strategyContext);
121:
122:         /// @notice params.minPrimary and params.minSecondary are not
 ↪  required to be passed in by the caller
123:         /// for this strategy vault
124:         (params.minPrimary, params.minSecondary) =
 ↪  poolContext.basePool._getMinExitAmounts({
125:             strategyContext: strategyContext,
126:             oraclePrice: oraclePrice,
127:             spotPrice: spotPrice,
128:             poolClaim: poolClaimToSettle
129:         });
```

The `TwoTokenPoolUtils._getMinExitAmounts` function calculates the minimum amount on the share of the pool with a small discount.

Assume a Curve Pool with the following configuration:

- Consist of two tokens (DAI and USDC). DAI is primary token, USDC is secondary token.
- Pool holds 200 US Dollars worth of tokens (50 DAI and 150 USDC).
- DAI <> USDC price is 1:1
- totalSupply = 100 LP Pool Tokens

Assume that 50 LP Pool Tokens will be claimed during vault settlement.

```
minPrimary = (poolContext.primaryBalance * poolClaim *
 ↪  strategyContext.vaultSettings.poolSlippageLimitPercent /  (totalPoolSupply *
 ↪  uint256(VaultConstants.VAULT_PERCENT_BASIS)
minPrimary = (50 DAI * 50 LP_TOKEN * 99.75% /  (100 LP_TOKEN * 100%)

Rewrite for clarity (ignoring rounding error):
minPrimary = 50 DAI * (50 LP_TOKEN/100 LP_TOKEN) * (99.75%/100%) = 24.9375 DAI

minSecondary = same calculation = 74.8125 USDC
```


`TwoTokenPoolUtils._getMinExitAmounts` function will return `24.9375 DAI` as `params.minPrimary` and `74.8125 USDC` as `params.minSecondary`.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/pool/TwoTokenPoolUtils.sol#L48

```
File: TwoTokenPoolUtils.sol
46:     /// @notice calculates the expected primary and secondary amounts based
 ↪  on
47:     /// the given spot price and oracle price
48:     function _getMinExitAmounts(
49:         TwoTokenPoolContext calldata poolContext,
50:         StrategyContext calldata strategyContext,
51:         uint256 spotPrice,
52:         uint256 oraclePrice,
53:         uint256 poolClaim
54:     ) internal view returns (uint256 minPrimary, uint256 minSecondary) {
55:         strategyContext._checkPriceLimit(oraclePrice, spotPrice);
56:
57:         // min amounts are calculated based on the share of the Balancer
 ↪  pool with a small discount applied
58:         uint256 totalPoolSupply = poolContext.poolToken.totalSupply();
59:         minPrimary = (poolContext.primaryBalance * poolClaim *
60:             strategyContext.vaultSettings.poolSlippageLimitPercent) /
61:             (totalPoolSupply * uint256(VaultConstants.VAULT_PERCENT_BASIS));
62:         minSecondary = (poolContext.secondaryBalance * poolClaim *
63:             strategyContext.vaultSettings.poolSlippageLimitPercent) /
64:             (totalPoolSupply * uint256(VaultConstants.VAULT_PERCENT_BASIS));
65:     }
```

When settling the vault, it is possible to instruct the vault to redeem the Curve's LP tokens single-sided or proportionally. Settle vault functions will trigger a chain of functions that will eventually call the `Curve2TokenConvexHelper._unstakeAndExitPool` function that is responsible for redeeming the Curve's LP tokens.

Within the `Curve2TokenConvexHelper._unstakeAndExitPool` function, if the `params.secondaryTradeParams.length` is zero, the redemption will be single-sided (refer to Line 242 below). Otherwise, the redemption will be executed proportionally (refer to Line 247 below). For a single-sided redemption, only the `params.minPrimary` will be used.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/internal/pool/Curve2TokenPoolUtils.sol#L231

```
File: Curve2TokenPoolUtils.sol
231:     function _unstakeAndExitPool(
232:         Curve2TokenPoolContext memory poolContext,
```

```
233:         ConvexStakingContext memory stakingContext,
234:         uint256 poolClaim,
235:         RedeemParams memory params
236:     ) internal returns (uint256 primaryBalance, uint256 secondaryBalance) {
237:         // Withdraw pool tokens back to the vault for redemption
238:         bool success =
↪   stakingContext.rewardPool.withdrawAndUnwrap(poolClaim, false); //
↪   claimRewards = false
239:         if (!success) revert Errors.UnstakeFailed();
240:
241:         if (params.secondaryTradeParams.length == 0) {
242:             // Redeem single-sided
243:             primaryBalance =
↪   ICurve2TokenPool(address(poolContext.curvePool)).remove_liquidity_one_coin(
244:                 poolClaim, int8(poolContext.basePool.primaryIndex),
↪   params.minPrimary
245:             );
246:         } else {
247:             // Redeem proportionally
248:             uint256[2] memory minAmounts;
249:             minAmounts[poolContext.basePool.primaryIndex] =
↪   params.minPrimary;
250:             minAmounts[poolContext.basePool.secondaryIndex] =
↪   params.minSecondary;
251:             uint256[2] memory exitBalances =
↪   ICurve2TokenPool(address(poolContext.curvePool)).remove_liquidity(
252:                 poolClaim, minAmounts
253:             );
254:
255:             (primaryBalance, secondaryBalance)
256:                 = (exitBalances[poolContext.basePool.primaryIndex],
↪   exitBalances[poolContext.basePool.secondaryIndex]);
257:         }
258:     }
```

Assume that the caller decided to perform a single-sided redemption of 50 LP Pool Tokens, using the earlier example. In this case,

- `poolClaim` = 50 LP Pool Tokens
- `params.minPrimary` = 24.9375 DAI
- `params.minSecondary` = 74.8125 USDC

The data passed into the `remove_liquidity_one_coin` will be as follows:

```
@notice Withdraw a single coin from the pool
@param _token_amount Amount of LP tokens to burn in the withdrawal
@param i Index value of the coin to withdraw
```

```
@param _min_amount Minimum amount of coin to receive
@return Amount of coin received
def remove_liquidity_one_coin(
    _token_amount: uint256,
    i: int128,
    _min_amount: uint256
) -> uint256:
```

```
remove_liquidity_one_coin(poolClaim, int8(poolContext.basePool.primaryIndex),
↪  params.minPrimary);
remove_liquidity_one_coin(50 LP_TOKEN, Index 0=DAI, 24.9375 DAI);
```

Assume the pool holds 200 US dollars worth of tokens (50 DAI and 150 USDC), and the total supply is 100 LP Tokens. The pool's state is imbalanced, so any trade will result in significant slippage.

Intuitively (ignoring the slippage & fee), redeeming 50 LP Tokens should return approximately 100 US dollars worth of tokens, which means around 100 DAI. Thus, the slippage or minimum amount should ideally be around 100 DAI (+/- 5%).

However, the trade will be executed in the above example even if the vault receives only 25 DAI because the `params.minPrimary` is set to `24.9375 DAI`. This could result in a loss of around 75 DAI due to slippage (about 75% slippage rate) in the worst-case scenario.

Impact

The slippage or minimum amount of tokens to be received is set to a value much smaller than expected. Thus, the vault will continue to redeem the pool tokens even if the trade incurs significant slippage, resulting in the vault receiving fewer tokens in return, leading to losses for the vault shareholders.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/external/Curve2TokenConvexHelper.sol#L112

Tool used

Manual Review

Recommendation

When performing a single-side redemption, avoid using the `TwoTokenPoolUtils._getMinExitAmounts` function to automatically compute the slippage or minimum amount of tokens to receive on behalf of the caller. Instead,

give the caller the flexibility to define the slippage (`params.minPrimary`). To prevent the caller from setting a slippage that is too large, consider restricting the slippage to an acceptable range.

The proper way of computing the minimum amount of tokens to receive from a single-side trade (`remove_liquidity_one_coin`) is to call the Curve Pool's calc_withdraw_one_coin function off-chain to calculate the amount received when withdrawing a single LP Token, and then apply an acceptable discount.

Note that the `calc_withdraw_one_coin` function cannot be used solely on-chain for computing the minimum amount because the result can be manipulated since it uses spot balances for computation.

Discussion

jeffywu

Valid, similar issue to #12 but for the remove_liquidity_one_coin method.

Issue H-5: Reinvest will return sub-optimal return if the pool is imbalanced

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/9

Found by

xiaoming90

Summary

Reinvesting only allows proportional deposit. If the pool is imbalanced due to unexpected circumstances, performing a proportional deposit is not optimal. This result in fewer pool tokens in return due to sub-optimal trade, eventually leading to a loss of gain for the vault shareholder.

Vulnerability Detail

During reinvest rewards, the vault will ensure that the amount of primary and secondary tokens deposited is of the right proportion per the comment in Line 163 below.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/external/Curve2TokenConvexHelper.sol#L163

```
File: Curve2TokenConvexHelper.sol
146:     function reinvestReward(
147:         Curve2TokenConvexStrategyContext calldata context,
148:         ReinvestRewardParams calldata params
149:     ) external {
..SNIP..
163:         // Make sure we are joining with the right proportion to minimize
 ↪  slippage
164:         poolContext._validateSpotPriceAndPairPrice({
165:             strategyContext: strategyContext,
166:             oraclePrice:
 ↪  poolContext.basePool._getOraclePairPrice(strategyContext),
167:             primaryAmount: primaryAmount,
168:             secondaryAmount: secondaryAmount
169:         });
```

The `Curve2TokenConvexHelper.reinvestReward` function will internally call the `Curve2TokenPoolUtils._checkPrimarySecondaryRatio`, which will check that the primary and secondary tokens deposited are of the right proportion.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/internal/pool/Curve2TokenPoolUtils.sol#L147

```
File: Curve2TokenPoolUtils.sol
147:     function _checkPrimarySecondaryRatio(
148:         StrategyContext memory strategyContext,
149:         uint256 primaryAmount,
150:         uint256 secondaryAmount,
151:         uint256 primaryPoolBalance,
152:         uint256 secondaryPoolBalance
153:     ) private pure {
154:         uint256 totalAmount = primaryAmount + secondaryAmount;
155:         uint256 totalPoolBalance = primaryPoolBalance +
↪  secondaryPoolBalance;
156:
157:         uint256 primaryPercentage = primaryAmount *
↪  CurveConstants.CURVE_PRECISION / totalAmount;
158:         uint256 expectedPrimaryPercentage = primaryPoolBalance *
↪  CurveConstants.CURVE_PRECISION / totalPoolBalance;
159:
160:         strategyContext._checkPriceLimit(expectedPrimaryPercentage,
↪  primaryPercentage);
161:
162:         uint256 secondaryPercentage = secondaryAmount *
↪  CurveConstants.CURVE_PRECISION / totalAmount;
163:         uint256 expectedSecondaryPercentage = secondaryPoolBalance *
↪  CurveConstants.CURVE_PRECISION / totalPoolBalance;
164:
165:         strategyContext._checkPriceLimit(expectedSecondaryPercentage,
↪  secondaryPercentage);
166:     }
```

This concept of proportional join appears to be taken from the design of earlier Notional's Balancer leverage vaults. For Balancer Pools, it is recommended to join with all the pool's tokens in exact proportions to minimize the price impact of the join (Reference).

However, the concept of proportional join to minimize slippage does not always hold for Curve Pools as they operate differently.

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 Curve Pool will always be balanced. Historically, there are multiple instances where the largest Curve pool (stETH/ETH) becomes imbalanced (Reference #1 and #2).

If the pool is imbalanced due to unexpected circumstances, performing a proportional deposit is not optimal, leading to the trade resulting in fewer tokens than possible due to the deposit penalty. In addition, the trade also misses out on the potential gain from the deposit bonus.

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

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/external/Curve2TokenConvexHelper.sol#L163

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/internal/pool/Curve2TokenPoolUtils.sol#L147

Tool used

Manual Review

Recommendation

Consider removing the `_checkPrimarySecondaryRatio` function from the `_validateSpotPriceAndPairPrice` function to give the callers the option to deposit the reward tokens in a "non-proportional" manner if a Curve Pool becomes imbalanced so that the deposit penalty could be minimized or the deposit bonus can be exploited to increase the return.

Discussion

jeffywu

Valid, should get the optimal way to join via some off chain function for Curve pools. This is will work because we are using a permissioned reward reinvestment role.

Issue H-6: Curve vault will undervalue or overvalue the LP Pool tokens if it comprises tokens with different decimals

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/8

Found by

xiaoming90

Summary

A Curve vault that comprises tokens with different decimals will undervalue or overvalue the LP Pool tokens. As a result, users might be liquidated prematurely or be able to borrow more than they are allowed. Additionally, the vault settlement process might break.

Vulnerability Detail

The `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function, which is utilized by the Curve vault, is used to compute the total value of the LP Pool tokens (`poolClaim`) denominated in the primary token.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/pool/TwoTokenPoolUtils.sol#L67

```
File: TwoTokenPoolUtils.sol
67:     function _getTimeWeightedPrimaryBalance(
68:         TwoTokenPoolContext memory poolContext,
69:         StrategyContext memory strategyContext,
70:         uint256 poolClaim,
71:         uint256 oraclePrice,
72:         uint256 spotPrice
73:     ) internal view returns (uint256 primaryAmount) {
74:         // Make sure spot price is within oracleDeviationLimit of pairPrice
75:         strategyContext._checkPriceLimit(oraclePrice, spotPrice);
76:
77:         // Get shares of primary and secondary balances with the provided
 ↪  poolClaim
78:         uint256 totalSupply = poolContext.poolToken.totalSupply();
79:         uint256 primaryBalance = poolContext.primaryBalance * poolClaim /
 ↪  totalSupply;
80:         uint256 secondaryBalance = poolContext.secondaryBalance * poolClaim
 ↪  / totalSupply;
81:
```

```
82:         // Value the secondary balance in terms of the primary token using
 ↪  the oraclePairPrice
83:         uint256 secondaryAmountInPrimary = secondaryBalance *
 ↪  strategyContext.poolClaimPrecision / oraclePrice;
84:
85:         // Make sure primaryAmount is reported in primaryPrecision
86:         uint256 primaryPrecision = 10 ** poolContext.primaryDecimals;
87:         primaryAmount = (primaryBalance + secondaryAmountInPrimary) *
 ↪  primaryPrecision / strategyContext.poolClaimPrecision;
88:     }
```

If a leverage vault supports a Curve Pool that contains two tokens with different decimals, the math within the `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function would not work, and the value returned from it will be incorrect. Consider the following two scenarios:

If primary token's decimals (e.g. 18) > secondary token's decimals (e.g. 6) To illustrate the issue, assume the following:

- The leverage vault supports the DAI-USDC Curve Pool, and its primary token of the vault is DAI.
- DAI's decimals are 18, while USDC's decimals are 6.
- Curve Pool's total supply is 100
- The Curve Pool holds 100 DAI and 100 USDC
- For the sake of simplicity, the price of DAI and USDC is 1:1. Thus, the `oraclePrice` within the function will be `1 * 10^18`. Note that the oracle price is always scaled up to 18 decimals within the vault.

The caller of the `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function wanted to compute the total value of 50 LP Pool tokens.

```
primaryBalance = poolContext.primaryBalance * poolClaim / totalSupply; // 100
 ↪  DAI * 50 / 100
secondaryBalance = poolContext.secondaryBalance * poolClaim / totalSupply; //
 ↪  100 USDC * 50 / 100
```

The `primaryBalance` will be `50 DAI`. `50 DAI` denominated in WEI will be `50 * 10^18` since the decimals of DAI are 18.

The `secondaryBalance` will be `50 USDC`. `50 USDC` denominated in WEI will be `50 * 10^6` since the decimals of USDC are 6.

Next, the code logic attempts to value the secondary balance (50 USDC) in terms of the primary token (DAI) using the oracle price (`1 * 10^18`).

```
secondaryAmountInPrimary = secondaryBalance * strategyContext.poolClaimPrecision
↪  / oraclePrice;
secondaryAmountInPrimary = 50 USDC * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = (50 * 10^6) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = 50 * 10^6
```

50 USDC should be worth 50 DAI (`50 * 10^18`). However, the `secondaryAmountInPrimary` shows that it is only worth 0.00000000005 DAI (`50 * 10^6`).

```
primaryAmount = (primaryBalance + secondaryAmountInPrimary) * primaryPrecision /
↪  strategyContext.poolClaimPrecision;
primaryAmount = [(50 * 10^18) + (50 * 10^6)] * 10^18 / 10^18
primaryAmount = [(50 * 10^18) + (50 * 10^6)] // cancel out the 10^18
primaryAmount = 50 DAI + 0.00000000005 DAI = 50.00000000005 DAI
```

50 LP Pool tokens should be worth 100 DAI. However, the `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function shows that it is only worth 50.00000000005 DAI, which undervalues the LP Pool tokens.

If primary token's decimals (e.g. 6) < secondary token's decimals (e.g. 18) To illustrate the issue, assume the following:

- The leverage vault supports the DAI-USDC Curve Pool, and its primary token of the vault is USDC.
- USDC's decimals are 6, while DAI's decimals are 18.
- Curve Pool's total supply is 100
- The Curve Pool holds 100 USDC and 100 DAI
- For the sake of simplicity, the price of DAI and USDC is 1:1. Thus, the `oraclePrice` within the function will be `1 * 10^18`. Note that the oracle price is always scaled up to 18 decimals within the vault.

The caller of the `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function wanted to compute the total value of 50 LP Pool tokens.

```
primaryBalance = poolContext.primaryBalance * poolClaim / totalSupply; // 100
↪  USDC * 50 / 100
secondaryBalance = poolContext.secondaryBalance * poolClaim / totalSupply; //
↪  100 DAI * 50 / 100
```

The `primaryBalance` will be `50 USDC`. `50 USDC` denominated in WEI will be `50 * 10^6` since the decimals of USDC are 6.

The `secondaryBalance` will be `50 DAI`. `50 DAI` denominated in WEI will be `50 * 10^18` since the decimals of DAI are 18.

Next, the code logic attempts to value the secondary balance (50 DAI) in terms of the primary token (USDC) using the oracle price (`1 * 10^18`).

```
secondaryAmountInPrimary = secondaryBalance * strategyContext.poolClaimPrecision
↪  / oraclePrice;
secondaryAmountInPrimary = 50 DAI * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = (50 * 10^18) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = 50 * 10^18
```

50 DAI should be worth 50 USDC (`50 * 10^6`). However, the `secondaryAmountInPrimary` shows that it is worth 50,000,000,000,000 USDC (`50 * 10^18`).

```
primaryAmount = (primaryBalance + secondaryAmountInPrimary) * primaryPrecision /
↪  strategyContext.poolClaimPrecision;
primaryAmount = [(50 * 10^6) + (50 * 10^18)] * 10^6 / 10^18
primaryAmount = [(50 * 10^6) + (50 * 10^18)] / 10^12
primaryAmount = 50,000,000.00005 = 50 million
```

50 LP Pool tokens should be worth 100 USDC. However, the `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function shows that it is worth 50 million USDC, which overvalues the LP Pool tokens.

In summary, if a leverage vault has two tokens with different decimals:

- If primary token's decimals (e.g. 18) > secondary token's decimals (e.g. 6), then `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function will undervalue the LP Pool tokens
- If primary token's decimals (e.g. 6) < secondary token's decimals (e.g. 18), then `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function will overvalue the LP Pool tokens

Impact

A vault supporting tokens with two different decimals will undervalue or overvalue the LP Pool tokens.

The affected `TwoTokenPoolUtils._getTimeWeightedPrimaryBalance` function is called within the `Curve2TokenPoolUtils._convertStrategyToUnderlying` function that is used for valuing strategy tokens in terms of the primary balance. As a result, the strategy tokens will be overvalued or undervalued

Following are some of the impacts of this issue:

- If the strategy tokens are overvalued or undervalued, the users might be liquidated prematurely or be able to borrow more than they are allowed to since the `Curve2TokenPoolUtils._convertStrategyToUnderlying` function is indirectly used for computing the collateral ratio of an account within Notional's `VaultConfiguration.calculateCollateralRatio` function.
- `expectedUnderlyingRedeemed` is computed based on the `Curve2TokenPoolUtils._convertStrategyToUnderlying` function. If the `expectedUnderlyingRedeemed` is incorrect, it will break the vault settlement process.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/pool/TwoTokenPoolUtils.sol#L67

Tool used

Manual Review

Recommendation

When valuing the secondary balance in terms of the primary token using the oracle price, the result should be scaled up or down the decimals of the primary token accordingly if the decimals of the two tokens are different.

The root cause of this issue is in the following portion of the code, which attempts to add the `primaryBalance` and `secondaryAmountInPrimary` before multiplying with the `primaryPrecision`. The `primaryBalance` and `secondaryAmountInPrimary` might not be denominated in the same decimals. Therefore, they cannot be added together without scaling them if the decimals of two tokens are different.

```
primaryAmount = (primaryBalance + secondaryAmountInPrimary) * primaryPrecision /
↪   strategyContext.poolClaimPrecision;
```

Consider implementing the following changes to ensure that the math within the `_getTimeWeightedPrimaryBalance` function work with tokens with different decimals. The below approach will scale the secondary token to match the primary token's precision before performing further computation.

```
function _getTimeWeightedPrimaryBalance(
    TwoTokenPoolContext memory poolContext,
    StrategyContext memory strategyContext,
    uint256 poolClaim,
    uint256 oraclePrice,
    uint256 spotPrice
) internal view returns (uint256 primaryAmount) {
```

```
    // Make sure spot price is within oracleDeviationLimit of pairPrice
    strategyContext._checkPriceLimit(oraclePrice, spotPrice);

    // Get shares of primary and secondary balances with the provided poolClaim
    uint256 totalSupply = poolContext.poolToken.totalSupply();
    uint256 primaryBalance = poolContext.primaryBalance * poolClaim /
↪   totalSupply;
    uint256 secondaryBalance = poolContext.secondaryBalance * poolClaim /
↪   totalSupply;

+   // Scale secondary balance to primaryPrecision
+   uint256 primaryPrecision = 10 ** poolContext.primaryDecimals;
+   uint256 secondaryPrecision = 10 ** poolContext.secondaryDecimals;
+   secondaryBalance = secondaryBalance * primaryPrecision / secondaryPrecision

    // Value the secondary balance in terms of the primary token using the
↪   oraclePairPrice
    uint256 secondaryAmountInPrimary = secondaryBalance *
↪   strategyContext.poolClaimPrecision / oraclePrice;

-   // Make sure primaryAmount is reported in primaryPrecision
-   uint256 primaryPrecision = 10 ** poolContext.primaryDecimals;
-   primaryAmount = (primaryBalance + secondaryAmountInPrimary) *
↪   primaryPrecision / strategyContext.poolClaimPrecision;
+   primaryAmount = primaryBalance + secondaryAmountInPrimary
}
```

The `poolContext.primaryBalance` or `poolClaim` are not scaled up to `strategyContext.poolClaimPrecision`. Thus, the `primaryBalance` is not scaled in any form. Thus, I do not see the need to perform any conversion at the last line of the `_getTimeWeightedPrimaryBalance` function.

```
uint256 primaryBalance = poolContext.primaryBalance * poolClaim / totalSupply;
```

The following attempts to run through the examples in the previous section showing that the updated function produces valid results after the changes.

If primary token's decimals (e.g. 18) > secondary token's decimals (e.g. 6)

```
Primary Balance = 50 DAI (18 Deci), Secondary Balance = 50 USDC (6 Deci)

secondaryBalance = secondaryBalance * primaryPrecision / secondaryPrecision
secondaryBalance = 50 USDC * 10^18 / 10^6
secondaryBalance = (50 * 10^6) * 10^18 / 10^6 = (50 * 10^18)

secondaryAmountInPrimary = secondaryBalance * strategyContext.poolClaimPrecision
↪   / oraclePrice;
```

```
secondaryAmountInPrimary = (50 * 10^18) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = (50 * 10^18) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = 50 * 10^18

primaryAmount = primaryBalance + secondaryAmountInPrimary
primaryAmount = (50 * 10^18) + (50 * 10^18) = (100 * 10^18) = 100 DAI
```

If primary token's decimals (e.g. 6) < secondary token's decimals (e.g. 18)

```
Primary Balance = 50 USDC (6 Deci), Secondary Balance = 50 DAI (18 Deci)

secondaryBalance = secondaryBalance * primaryPrecision / secondaryPrecision
secondaryBalance = 50 DAI * 10^6 / 10^18
secondaryBalance = (50 * 10^18) * 10^6 / 10^18 = (50 * 10^6)

secondaryAmountInPrimary = secondaryBalance * strategyContext.poolClaimPrecision
↪  / oraclePrice;
secondaryAmountInPrimary = (50 * 10^6) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = (50 * 10^6) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = 50 * 10^6

primaryAmount = primaryBalance + secondaryAmountInPrimary
primaryAmount = (50 * 10^6) + (50 * 10^6) = (100 * 10^6) = 100 USDC
```

If primary token's decimals (e.g. 6) == secondary token's decimals (e.g. 6)

```
Primary Balance = 50 USDC (6 Deci), Secondary Balance = 50 USDT (6 Deci)

secondaryBalance = secondaryBalance * primaryPrecision / secondaryPrecision
secondaryBalance = 50 USDT * 10^6 / 10^6
secondaryBalance = (50 * 10^6) * 10^6 / 10^6 = (50 * 10^6)

secondaryAmountInPrimary = secondaryBalance * strategyContext.poolClaimPrecision
↪  / oraclePrice;
secondaryAmountInPrimary = (50 * 10^6) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = (50 * 10^6) * 10^18 / (1 * 10^18)
secondaryAmountInPrimary = 50 * 10^6

primaryAmount = primaryBalance + secondaryAmountInPrimary
primaryAmount = (50 * 10^6) + (50 * 10^6) = (100 * 10^6) = 100 USDC
```

`strategyContext.poolClaimPrecision` set to `CurveConstants.CURVE_PRECISION`, which is `1e18`. `oraclePrice` is always in `1e18` precision.

Discussion

jeffywu

Valid, since this is shared code it is better to always scale decimals up to 18 rather than assume that they are. Even if this results in duplicate work for the Balancer strategies.

Issue M-1: Vault cannot be deployed properly for newer Curve pools

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/22

Found by

usmannk

Summary

New Curve pools have a different ABI than the one that the vaults attempt to use. Not differentiating between the two will lead to silent failures when vaults are deployed for new pools.

Vulnerability Detail

During the deploy process the leveraged vault attempts to obtain the Curve pool's LP token address by calling `CURVE_POOL.lp_token()`. However, for new pools such as CRV/ETH (https://etherscan.io/address/0x8301ae4fc9c624d1d396cbdaa1ed877821d7c511#code), there is no `lp_token` member. Instead calling `lp_token()` will trigger the fallback function. For these pools, the token is obtained by calling `token()`.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/mixins/CurvePoolMixin.sol#L25

Impact

The lp token's address will be set to the 0 address and later calls to functions such as `totalSupply` will fail, causing the vault to be in a stuck state.

Code Snippet

Tool used

Manual Review

Recommendation

Check the version of the target Curve pool and toggle called functions appropiately.

Discussion

jeffywu

Valid

Issue M-2: `oracleSlippagePercentOrLimit` can exceed the `Constants.SLIPPAGE_LIMIT_PRECISION`

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/19

Found by

xiaoming90

Summary

Trade might be settled with a large slippage causing a loss of assets as the `oracleSlippagePercentOrLimit` limit is not bounded and can exceed the `Constants.SLIPPAGE_LIMIT_PRECISION` threshold.

Vulnerability Detail

The code at Line 73-75 only checks if the `oracleSlippagePercentOrLimit` is within the `Constants.SLIPPAGE_LIMIT_PRECISION` if `useDynamicSlippage` is `true`. If the trade is performed without dynamic slippage, the trade can be executed with an arbitrary limit.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/strategy/StrategyUtils.sol#L62

```
File: StrategyUtils.sol
62:     function _executeTradeExactIn(
63:         TradeParams memory params,
64:         ITradingModule tradingModule,
65:         address sellToken,
66:         address buyToken,
67:         uint256 amount,
68:         bool useDynamicSlippage
69:     ) internal returns (uint256 amountSold, uint256 amountBought) {
70:         require(
71:             params.tradeType == TradeType.EXACT_IN_SINGLE ||
 ↪  params.tradeType == TradeType.EXACT_IN_BATCH
72:         );
73:         if (useDynamicSlippage) {
74:             require(params.oracleSlippagePercentOrLimit <=
 ↪  Constants.SLIPPAGE_LIMIT_PRECISION);
75:         }
76:
77:         // Sell residual secondary balance
78:         Trade memory trade = Trade(
79:             params.tradeType,
```

```
80:             sellToken,
81:             buyToken,
82:             amount,
83:             useDynamicSlippage ? 0 : params.oracleSlippagePercentOrLimit,
84:             block.timestamp, // deadline
85:             params.exchangeData
86:         );
```

The `StrategyUtils._executeTradeExactIn` function is utilized by the Curve Vault.

Impact

Trade might be settled with a large slippage causing a loss of assets.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/strategy/StrategyUtils.sol#L62

Tool used

Manual Review

Recommendation

Consider restricting the slippage limit when a trade is executed without dynamic slippage.

```
    function _executeTradeExactIn(
        TradeParams memory params,
        ITradingModule tradingModule,
        address sellToken,
        address buyToken,
        uint256 amount,
        bool useDynamicSlippage
    ) internal returns (uint256 amountSold, uint256 amountBought) {
        require(
            params.tradeType == TradeType.EXACT_IN_SINGLE || params.tradeType ==
↪   TradeType.EXACT_IN_BATCH
        );
        if (useDynamicSlippage) {
            require(params.oracleSlippagePercentOrLimit <=
↪   Constants.SLIPPAGE_LIMIT_PRECISION);
-       }
+       } else {
```

```
+           require(params.oracleSlippagePercentOrLimit != 0 &&
 ↪  params.oracleSlippagePercentOrLimit <=
 ↪  Constants.SLIPPAGE_LIMIT_PRECISION_FOR_NON_DYNAMIC_TRADE);
+       }
```

Discussion

jeffywu

Valid, I agree with the intent in this issue although I'm not sure if the proposed solution is workable. We don't have guardrails against this method being called with dynamicSlippage set to false by a non-authenticated account from an internal code perspective. We should consider some sort of internal check to ensure that this is the case.

Issue M-3: Oracle slippage rate is used for checking primary and secondary ratio

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/18

Found by

xiaoming90

Summary

The oracle slippage rate (`oraclePriceDeviationLimitPercent`) is used for checking the ratio of the primary and secondary tokens to be deposited into the pool.

As a result, changing the `oraclePriceDeviationLimitPercent` setting to increase or decrease the allowable slippage between the spot and oracle prices can cause unexpected side-effects to the `_checkPrimarySecondaryRatio` function, which might break the `reinvestReward` function that relies on the `_checkPrimarySecondaryRatio` function under certain condition.

Vulnerability Detail

The `_checkPriceLimit` function is for the purpose of comparing the spot price with the oracle price. Thus, the slippage (`oraclePriceDeviationLimitPercent`) is specially selected for this purpose.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/strategy/StrategyUtils.sol#L21

```
File: StrategyUtils.sol
21:     function _checkPriceLimit(
22:         StrategyContext memory strategyContext,
23:         uint256 oraclePrice,
24:         uint256 poolPrice
25:     ) internal pure {
26:         uint256 lowerLimit = (oraclePrice *
27:             (VaultConstants.VAULT_PERCENT_BASIS -
↪  strategyContext.vaultSettings.oraclePriceDeviationLimitPercent)) /
28:             VaultConstants.VAULT_PERCENT_BASIS;
29:         uint256 upperLimit = (oraclePrice *
30:             (VaultConstants.VAULT_PERCENT_BASIS +
↪  strategyContext.vaultSettings.oraclePriceDeviationLimitPercent)) /
31:             VaultConstants.VAULT_PERCENT_BASIS;
32:
33:         if (poolPrice < lowerLimit || upperLimit < poolPrice) {
34:             revert Errors.InvalidPrice(oraclePrice, poolPrice);
```

```
35:         }
36:     }
```

However, it was observed that `_checkPriceLimit` function is repurposed for checking if the ratio of the primary and secondary tokens to be deposited to the pool is more or less proportional to the pool's balances within the `_checkPrimarySecondaryRatio` function during reinvestment.

The `oraclePriceDeviationLimitPercent` setting should not be used here as it does not involve any oracle data. Thus, the correct way is to define another setting specifically for checking if the ratio of the primary and secondary tokens to be deposited to the pool is more or less proportional to the pool's balances.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/internal/pool/Curve2TokenPoolUtils.sol#L147

```
File: Curve2TokenPoolUtils.sol
147:     function _checkPrimarySecondaryRatio(
148:         StrategyContext memory strategyContext,
149:         uint256 primaryAmount,
150:         uint256 secondaryAmount,
151:         uint256 primaryPoolBalance,
152:         uint256 secondaryPoolBalance
153:     ) private pure {
154:         uint256 totalAmount = primaryAmount + secondaryAmount;
155:         uint256 totalPoolBalance = primaryPoolBalance +
↪  secondaryPoolBalance;
156:
157:         uint256 primaryPercentage = primaryAmount *
↪  CurveConstants.CURVE_PRECISION / totalAmount;
158:         uint256 expectedPrimaryPercentage = primaryPoolBalance *
↪  CurveConstants.CURVE_PRECISION / totalPoolBalance;
159:
160:         strategyContext._checkPriceLimit(expectedPrimaryPercentage,
↪  primaryPercentage);
161:
162:         uint256 secondaryPercentage = secondaryAmount *
↪  CurveConstants.CURVE_PRECISION / totalAmount;
163:         uint256 expectedSecondaryPercentage = secondaryPoolBalance *
↪  CurveConstants.CURVE_PRECISION / totalPoolBalance;
164:
165:         strategyContext._checkPriceLimit(expectedSecondaryPercentage,
↪  secondaryPercentage);
166:     }
```

Impact

Changing the `oraclePriceDeviationLimitPercent` setting to increase or decrease the allowable slippage between the spot price and oracle price can cause unexpected side-effects to the `_checkPrimarySecondaryRatio` function, which might break the `reinvestReward` function that relies on the `_checkPrimarySecondaryRatio` function under certain condition.

Additionally, the value chosen for the `oraclePriceDeviationLimitPercent` is to compare the spot price with the oracle price. Thus, it might not be the optimal value for checking if the ratio of the primary and secondary tokens deposited to the pool is more or less proportional to the pool's balances.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/common/internal/strategy/StrategyUtils.sol#L21

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/internal/pool/Curve2TokenPoolUtils.sol#L147

Tool used

Manual Review

Recommendation

There is a difference between the slippage for the following two items:

- Allowable slippage between the spot price and oracle price
- Allowable slippage between the ratio of the primary and secondary tokens to be deposited to the pool against the pool's balances

Since they serve a different purposes, they should not share the same slippage. Consider defining a separate slippage setting and function for checking if the ratio of the primary and secondary tokens deposited to the pool is more or less proportional to the pool's balances.

Discussion

jeffywu

Valid, should add a second setting here.

Issue M-4: Logic Error due to different representation of Native ETH (0x0 & 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeee

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/15

Found by

xiaoming90

Summary

Unexpected results might occur during vault initialization if either of the pool's tokens is a Native ETH due to the confusion between `Deployments.ETH_ADDRESS (address(0))` and `Deployments.ALT_ETH_ADDRESS (0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeee)`.

Vulnerability Detail

The `PRIMARY_TOKEN` or `SECONDARY_TOKEN` is explicitly converted to `Deployments.ETH_ADDRESS (address(0)` during deployment.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/curve/mixins/Curve2TokenPoolMixin.sol#L24

```
File: Curve2TokenPoolMixin.sol
abstract contract Curve2TokenPoolMixin is CurvePoolMixin {
..SNIP..
24:     constructor(
25:         NotionalProxy notional_,
26:         ConvexVaultDeploymentParams memory params
27:     ) CurvePoolMixin(notional_, params) {
28:         address primaryToken =
↪   _getNotionalUnderlyingToken(params.baseParams.primaryBorrowCurrencyId);
29:
30:         PRIMARY_TOKEN = primaryToken;
31:
32:         // Curve uses ALT_ETH_ADDRESS
33:         if (primaryToken == Deployments.ETH_ADDRESS) {
34:             primaryToken = Deployments.ALT_ETH_ADDRESS;
35:         }
36:
37:         address token0 = CURVE_POOL.coins(0);
38:         address token1 = CURVE_POOL.coins(1);
39:
40:         uint8 primaryIndex;
41:         address secondaryToken;
```

```
42:         if (token0 == primaryToken) {
43:             primaryIndex = 0;
44:             secondaryToken = token1;
45:         } else {
46:             primaryIndex = 1;
47:             secondaryToken = token0;
48:         }
49:
50:         if (secondaryToken == Deployments.ALT_ETH_ADDRESS) {
51:             secondaryToken = Deployments.ETH_ADDRESS;
52:         }
53:
54:         PRIMARY_INDEX = primaryIndex;
55:         SECONDARY_TOKEN = secondaryToken;
```

It was observed that there is a logic error within the `Curve2TokenConvexVault.initialize` function. Based on Lines 56 and 59 within the `Curve2TokenConvexVault.initialize` function, it assumes that if either the primary or secondary token is ETH, then the `PRIMARY_TOKEN` or `SECONDARY_TOKEN` will be set to `Deployments.ALT_ETH_ADDRESS`, which point to `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`.

However, this is incorrect as the `PRIMARY_TOKEN` or `SECONDARY_TOKEN` has already been converted to `Deployments.ETH_ADDRESS (address(0))` during deployment. Refer to the constructor of `Curve2TokenPoolMixin`.

Thus, the `PRIMARY_TOKEN` or `SECONDARY_TOKEN` will never be equal to `Deployments.ALT_ETH_ADDRESS`, and the condition at Lines 56 and 59 will always evaluate to True.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol#L48

```
File: Curve2TokenConvexVault.sol
contract Curve2TokenConvexVault is Curve2TokenVaultMixin {
..SNIP..
48:     function initialize(InitParams calldata params)
49:         external
50:         initializer
51:         onlyNotionalOwner
52:     {
53:         __INIT_VAULT(params.name, params.borrowCurrencyId);
54:         CurveVaultStorage.setStrategyVaultSettings(params.settings);
55:
56:         if (PRIMARY_TOKEN != Deployments.ALT_ETH_ADDRESS) {
57:             IERC20(PRIMARY_TOKEN).checkApprove(address(CURVE_POOL),
 ↪  type(uint256).max);
58:         }
```

```
59:         if (SECONDARY_TOKEN != Deployments.ALT_ETH_ADDRESS) {
60:             IERC20(SECONDARY_TOKEN).checkApprove(address(CURVE_POOL),
 ↪  type(uint256).max);
61:         }
62:
63:         CURVE_POOL_TOKEN.checkApprove(address(CONVEX_BOOSTER),
 ↪  type(uint256).max);
64:     }
```

As a result, if the `PRIMARY_TOKEN` or `SECONDARY_TOKEN` is `Deployments.ETH_ADDRESS` (`address(0)`), the code will go ahead to call the `checkApprove` function, which might cause unexpected results during vault initialization.

Impact

Unexpected results during vault initialization if either of the pool's tokens is a Native ETH.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/vaults/Curve2TokenConvexVault.sol#L48

Tool used

Manual Review

Recommendation

If the `PRIMARY_TOKEN` or `SECONDARY_TOKEN` is equal to `Deployments.ALT_ETH_ADDRESS` or `Deployments.ETH_ADDRESS`, this means that it points to native ETH and the `checkApprove` can be safely skipped.

```
function initialize(InitParams calldata params)
    external
    initializer
    onlyNotionalOwner
{
    __INIT_VAULT(params.name, params.borrowCurrencyId);
    CurveVaultStorage.setStrategyVaultSettings(params.settings);

-   if (PRIMARY_TOKEN != Deployments.ALT_ETH_ADDRESS) {
+   if (PRIMARY_TOKEN != Deployments.ALT_ETH_ADDRESS || PRIMARY_TOKEN !=
 ↪  Deployments.ETH_ADDRESS) {
        IERC20(PRIMARY_TOKEN).checkApprove(address(CURVE_POOL),
 ↪  type(uint256).max);
```

```
    }
-   if (SECONDARY_TOKEN != Deployments.ALT_ETH_ADDRESS) {
+   if (SECONDARY_TOKEN != Deployments.ALT_ETH_ADDRESS || SECONDARY_TOKEN !=
↪   Deployments.ETH_ADDRESS) {
        IERC20(SECONDARY_TOKEN).checkApprove(address(CURVE_POOL),
↪   type(uint256).max);
    }

    CURVE_POOL_TOKEN.checkApprove(address(CONVEX_BOOSTER), type(uint256).max);
}
```

Discussion

jeffywu

This issue appears valid, however, it's curious that this was not caught in unit tests since we tested against the wstETH/ETH pool which should have tripped this issue.

We can mark it as valid, @weitianjie2000 will investigate more on why the unit tests worked.

Issue M-5: Users are forced to use the first pool returned by the Curve Registry

Source: https://github.com/sherlock-audit/2023-02-notional-judging/issues/11

Found by

xiaoming90

Summary

If multiple pools support the exchange, users are forced to use the first pool returned by the Curve Registry. The first pool returned by Curve Registry might not be the most optimal pool to trade with. The first pool might have lesser liquidity, larger slippage, and higher fee than the other pools, resulting in the trade returning lesser assets than expected.

Vulnerability Detail

When performing a trade via the `CurveAdapter._exactInSingle` function, it will call the CURVE_REGISTRY.find_pool_for_coins function to find the available pools for exchanging two coins.

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/trading/adapters/CurveAdapter.sol#L34

```
File: CurveAdapter.sol
34:     function _exactInSingle(Trade memory trade)
35:         internal view returns (address target, bytes memory
 ↪  executionCallData)
36:     {
37:         address sellToken = _getTokenAddress(trade.sellToken);
38:         address buyToken = _getTokenAddress(trade.buyToken);
39:         ICurvePool pool =
 ↪  ICurvePool(Deployments.CURVE_REGISTRY.find_pool_for_coins(sellToken,
 ↪  buyToken));
40:
41:         if (address(pool) == address(0)) revert InvalidTrade();
42:
43:         int128 i = -1;
44:         int128 j = -1;
45:         for (int128 c = 0; c < MAX_TOKENS; c++) {
46:             address coin = pool.coins(uint256(int256(c)));
47:             if (coin == sellToken) i = c;
48:             if (coin == buyToken) j = c;
49:             if (i > -1 && j > -1) break;
```

```
50:         }
51:
52:         if (i == -1 || j == -1) revert InvalidTrade();
53:
54:         return (
55:             address(pool),
56:             abi.encodeWithSelector(
57:                 ICurvePool.exchange.selector,
58:                 i,
59:                 j,
60:                 trade.amount,
61:                 trade.limit
62:             )
63:         );
64:     }
```

However, it was observed that when multiple pools are available, users can choose the pool to return by defining the `i` parameter of the `find_pool_for_coins` function as shown below.

https://etherscan.io/address/0x90E00ACe148ca3b23Ac1bC8C240C2a7Dd9c2d7f5#code

```
@view
@external
def find_pool_for_coins(_from: address, _to: address, i: uint256 = 0) -> address:
    """
    @notice Find an available pool for exchanging two coins
    @param _from Address of coin to be sent
    @param _to Address of coin to be received
    @param i Index value. When multiple pools are available
            this value is used to return the n'th address.
    @return Pool address
    """
    key: uint256 = bitwise_xor(convert(_from, uint256), convert(_to, uint256))
    return self.markets[key][i]
```

However, the `CurveAdapter._exactInSingle` did not allow users to define the `i` parameter of the `find_pool_for_coins` function. As a result, users are forced to trade against the first pool returned by the Curve Registry.

Impact

The first pool returned by Curve Registry might not be the most optimal pool to trade with. The first pool might have lesser liquidity, larger slippage, and higher fee than the other pools, resulting in the trade returning lesser assets than expected.

Code Snippet

https://github.com/sherlock-audit/2023-02-notional/blob/main/leveraged-vaults/contracts/trading/adapters/CurveAdapter.sol#L34

Tool used

Manual Review

Recommendation

If multiple pools support the exchange, consider allowing the users to choose which pool they want to trade against.

```
function _exactInSingle(Trade memory trade)
    internal view returns (address target, bytes memory executionCallData)
{
    address sellToken = _getTokenAddress(trade.sellToken);
    address buyToken = _getTokenAddress(trade.buyToken);
-   ICurvePool pool =
 ↪  ICurvePool(Deployments.CURVE_REGISTRY.find_pool_for_coins(sellToken,
 ↪  buyToken));
+   ICurvePool pool =
 ↪  ICurvePool(Deployments.CURVE_REGISTRY.find_pool_for_coins(sellToken,
 ↪  buyToken, trade.pool_index));
```

Discussion

jeffywu

Valid

hrishibhat

Given that this is only a possibility and all the conditions for loss of funds are not guaranteed, Considering this issue as a valid medium