certora

Security Assessment
Final Report

Underscore Vaults

November 2025

Prepared for Hightop

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Underscore	https://github.com/underscore-finance/underscore-protocol.git	a5235b4 (Earn Vaults Commit Hash) 68473e8 (fix review final commit)	Vyper
Underscore	https://github.com/underscore-finance/underscore-protocol/pull/34	9d34e89 (Leverage Vaults Commit Hash) 68473e8 (fix review final commit)	Vyper

Project Overview

This document describes the manual code review findings of **Underscore**. The following contract list is included in our scope:

```
contracts/vaults/EarnVault.vy
contracts/vaults/LevgVault.vy
contracts/vaults/LevgVaultHelper.vy
contracts/vaults/modules/EarnVaultWallet.vy
contracts/vaults/modules/LevgVaultWallet.vy
contracts/vaults/modules/VaultErc20Token.vy
contracts/registries/VaultRegistry.vy
contracts/data/Ledger.vy
contracts/data/MissionControl.vy
contracts/modules/YieldLegoData.vy
```

The work was undertaken from **October 13, 2025,** to **November 17, 2025**. During this time, Certora's security researchers performed a manual audit of all the Vyper contracts and discovered several bugs in the codebase, which are summarized in the subsequent section.

Protocol Overview

The Underscore Protocol vault system wraps complex, cross-protocol yield strategies into standard ERC-4626 vaults. At the base layer are Earn Vaults., where users deposit an underlying asset such as USDC or WETH and receive shares backed by a basket of yield positions. The EarnVaultWallet module tracks balances across approved external vault tokens, computes total underlying assets and realized yield, and uses Lego adapters to deposit, withdraw, swap, and claim rewards according to rules stored in the designated vault registry. Managers or AI agents operate the vault through these adapters, but their actions are always checked against the current configuration and `MissionControl` permissions.

Leveraged Vaults sit on top of this layer and build leveraged strategies using Earn Vaults themselves as both collateral and leverage assets. A `LevgVault` accepts deposits in a base asset, then routes that capital through `LevgVaultWallet` and `LevgVaultHelper` into specific Earn Vault tokens that represent the collateral side and the borrowed-side yield positions. The contracts enforce risk controls such as a maximum debt ratio, slippage limits on GREEN and USDC flows, and structured redemption logic that can deleverage and unwind positions from Ripe and Earn Vaults before fulfilling withdrawals. As with Earn Vaults, `LevgVaults` read configuration and safety flags from VaultRegistry, rely on the same manager and agent permissioning stack, and expose a simple ERC-4626 interface to users while hiding the underlying multi-protocol mechanics.

Release Readiness - Disclaimer

Following an internal review, we believe that although the security of the protocol has been increased, the current implementation still presents a security risk.

There are three main factors contributing to this assessment:

- The volume and severity of findings to date, spread between the 2 system components - vaults and wallets, which is concerning for a release candidate.
- Ongoing development activity, including feature changes and refactoring during the audit period, outside of fixes, which has introduced new, unaudited code paths that may carry additional risk.
- Integrations with external protocols and routing to them, which cannot be fully delved into given the time-boxed review and the sheer volume of code.

Given this, with transparency in mind - we would not feel comfortable recommending that the current codebase be taken to production right after the conclusion of this audit. In our view, the protocol would still require an additional review to reach a launch-ready state, considering the amount of complex and moving parts and its volume of components and external integrations.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	5	5	4
Medium	3	3	2
Low	4	4	2
Informational	2	2	0
Total	14	14	8

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
H-01	Wrong lego ids can be maliciously or accidentally registered	High	Fixed
H-02	Withdraw/redeem use different asset calculations	High	Fixed
H-03	Leverage vault calculate underlying Earn vault assets using max amounts only	High	Fixed
H-04	While redeeming from Earn vaults, users can receive 10% less assets than intended while burning all the shares	High	Fixed
H-05	First depositor's funds can be stolen through a vault inflation attack	High	Acknowledged
M-01	Manager can steal the majority of vault assets by re-depositing vault tokens	Medium	Fixed
M-02	Managers can swap out to a token which isn't the vault token by using phantom legos	Medium	Fixed

M-03	Calculation of maxBorrowAmount in USDC vaults is flawed	Medium	Acknowledged
L-01	Weighted price per share can deviate from the true recent	Low	Acknowledged
L-02	Clearing a vault token opportunity does not clear its snapshots	Low	Fixed
L-03	Vault Wallet manager indexes are not properly cleared and could cause confusion	Low	Fixed
L-04	If the maxNumSnapshots gets reduced, the next snapshot write will be written out of bounds	Low	Acknowledged
I-01	Collateral/Leverage vault changes can be bricked via 1 wei donations	Informational	Acknowledged
I-02	Blacklisted addresses can still redeem vault token for assets	Informational	Acknowledged

High Severity Issues

H-01. Wrong lego ids can be maliciously or accidentally registered

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: legos/yield/*	Status: Fixed	

Description:

All independent Lego contracts serve as the vaults and wallets' entry point to the external protocols they interact with. However they are also completely permissionless to be used by external users as router-like interfaces. The problem here is that global and local Underscore state is managed partially, via adding assets/vault tokens to the ledger.

The main struct that is used to manage the critical addresses, the `miniAddys` is completely user provided and could be crafted in such a way that we can register a lego id pointing to a completely different address:

```
Python
   def depositForYield(
    _asset: address,
    _amount: uint256,
    _vaultAddr: address,
    _extraData: bytes32,
    _recipient: address,
    _miniAddys: ws.MiniAddys = empty(ws.MiniAddys),
) -> (uint256, address, uint256, uint256):
    assert not yld.isPaused # dev: paused
    miniAddys: ws.MiniAddys = yld._getMiniAddys(_miniAddys)
    vaultInfo: ls.VaultTokenInfo = self._getVaultInfoOnDeposit(_asset, _vaultAddr,
miniAddys.ledger, miniAddys.legoBook)

......

@internal
```

```
def _registerVaultTokenGlobally(_underlyingAsset: address, _vaultToken: address, _decimals:
uint256, _ledger: address, _legoBook: address):
    if not staticcall Ledger(_ledger).isRegisteredVaultToken(_vaultToken):
        legoId: uint256 = staticcall Registry(_legoBook).getRegId(self)
        extcall Ledger(_ledger).setVaultToken(_vaultToken, legoId, _underlyingAsset,
_decimals, self._isRebasing())
```

As it can be seen, the addys addresses, such as the lego book used when registering to the ledger are user provided and lack sufficient validation that they match the protocol's intended addresses.

The deposits itself pass successfully, but we completely skew with every contract that is somehow dependent on the correct lego address being attached to the id, currently:

`Appraiser`, `Billing`, `Hatchery`, `LootDistributor`, `VaultRegistry`

Recommendations: Verify the provided addresses and overall critical variables used in the legos and consider completely limited access to them, since their ability to write to the global state of the protocol is too high risk

Customer's response: Fixed in 309f216

Fix Review: Fix confirmed

H-02. Withdraw/redeem use different asset calculations

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: EarnVault.vy	Status: Fixed	

Description:

There's a difference in asset calculations between `withdraw` and `redeem`, in which one of the functions would yield higher tokens for the caller.

The withdrawal path in the vault uses the maximized version of the totalAssets/totalBalance, while it should be using the safe one. This is evident from the call to the `_getUnderlyingData` with the first argument set to `true`.

Python

```
totalAssets, currentBalance, pendingYieldRealized, maxBalVaultToken =
self._getUnderlyingData(True, vaultRegistry)
shares: uint256 = self._amountToShares(_assets, token.totalSupply, totalAssets, True)
```

Considering this, the call to `_amountToShares` would have a larger denominator due to `totalBalance` being the larger value, thus we burn less shares per asset

The opposite happens during redemption inside `sharesToAmount`, since the total balance there uses the safe route and is thus lower, we would redeem less assets for burning the same amount of shares.

Recommendations: Withdraw should also be using the safe asset amount, and set the first argument to the `_getUnderlyingData` call to false.

Customer's response: Fixed in 1bd3736

Fix Review: Fix confirmed

H-03. Leverage vault calculates underlying Earn vault assets using max amounts only

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: LevgVault.vy	Status: Fixed	

Description:

The leverage vault share price always uses the max value in order to calculate the total underlying assets:

```python
def _getUnderlyingForVaultToken(
    _wallet: address,
    _vaultToken: address,
    _vaultTokenLegoId: uint256,
    _ripeVaultId: uint256,
    _legoBook: address,
    _ripeVaultBook: address,
) -> uint256:
    if _vaultToken == empty(address):
        return 0

    # vault token local balance
    vaultTokenAmount: uint256 = staticcall IERC20(_vaultToken).balanceOf(_wallet)

    # vault token on ripe protocol
    if _ripeVaultId != 0:
        ripeDepositAddr: address = staticcall Registry(_ripeVaultBook).getAddr(_ripeVaultId)
        if ripeDepositAddr != empty(address):
            vaultTokenAmount += self._getCollateralBalance(_wallet, _vaultToken,
ripeDepositAddr)

    if vaultTokenAmount == 0:
        return 0

    # calc underlying amount
```

```
    underlyingAmount: uint256 = 0
    legoAddr: address = staticcall Registry(_legoBook).getAddr(_vaultTokenLegoId)
    if legoAddr != empty(address):
        underlyingAmount = staticcall YieldLego(legoAddr).getUnderlyingAmount(_vaultToken,
vaultTokenAmount)

    return underlyingAmount
```

This is evident from the getUnderlyingAmount call to the Underscore lego. This call further invokes the `convertToAssets` call on the `EarnVault` which by default uses its max price. This happens when determining the total underlying on both deposits and withdrawals. However, the redemption mechanism behind withdrawals WILL use EarnVault's safe price, creating a discrepancy between the 2 mechanisms.

When the leverage vault is entered, the share price is calculated using totalAssets which also take into consideration assets deposited to the Earn vault, these are calculated using convertToAssets which uses Earn vault's max amounts. (which is correct).

The problem is during redeems, the totalAssets (used for share calculations) are also again calculated using convertToAssets which will use the max amounts, but if those assets were to be redeemed from the earn vault, they would be using the safeAssets param, not the max one, which is usually lower.

This leads to a couple of things, first the leverage vault would be at a "loss" since this won't consider the safePrice param when calculating shares/assets and will over-calculate them during redeems.

This difference will be socialized only by the last depositor(s) as all others would get higher assets for their shares, while the last one will socialize all the difference.

Recommendations: Correctly use the safe mechanism when doing withdrawals on the leverage vaults as well.

Customer's response: Fixed in 201baff

Fix Review: Fix confirmed

H-04. While redeeming from Earn vaults, users can receive 10% less assets than intended while burning all the shares

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: EarnVault.vy	Status: Fixed	

Description:

During redeems, the intended design of the mechanism is that users can redeem up to 0.1% less assets than intended, (0.1% tolerance) if there's some kind of an asset shortage in the underlying vaults, or if for some reason less assets are redeemed from the underlying vaults.

The problem is that with the current implementation, the tolerance isn't 0.1%, but it's actually 10%, allowing for users to withdraw 10% less assets than what they're supposed to according to the current share price, while burning all of the shares.

Python

```
 availAmount, withdrawnAmount = vaultWallet._prepareRedemption(_asset, _amount,
_maxBalVaultToken, _sender, _vaultRegistry)
    actualAmount: uint256 = min(availAmount, _amount)
    assert actualAmount >= _amount - (_amount // 10) # dev: insufficient funds (0.1%
tolerance)

    # save data
    currentBalance: uint256 = _currentBalance - min(_currentBalance, withdrawnAmount)
    vaultWallet.lastUnderlyingBal = currentBalance
    vaultWallet.pendingYieldRealized = _pendingYieldRealized

    # burn shares, transfer assets
    token._burn(_owner, _shares)
    assert extcall IERC20(_asset).transfer(_recipient, actualAmount,
default_return_value=True) # dev: withdrawal failed
```

As it can be seen from above, the amount isn't multiplied by 10, and then divided by 100_00 (which represents 100% within the current system), but it's just divided by 10.

When dividing the intended amount by 10, and then subtracting it from the original one, we're allowing for the actual amount to be up to 10% less than the intended one.

Recommendations: Multiply the amount by 10, and then divide it by 100_00 in order to get 0.1%.

Customer's response: Fixed in 29bdbdc

Fix Review: Fix confirmed

H-05. First depositor's funds can be stolen through a vault inflation attack

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: EarnVault.vy	Status: Acknowledged	

Description:

Due to the current design of the vaults, and its reliance on balanceOf in order to get the vault asset amount, a classical vault inflation attack can be performed in order to steal the majority of the first depositor's funds.

Python

```
def _amountToShares(
    _amount: uint256,
    _totalShares: uint256,
    _totalBalance: uint256,
    _shouldRoundUp: bool,
) -> uint256:
    if _amount == max_value(uint256) or _amount == 0:
        return _amount

    # first deposit, price per share = 1
    if _totalShares == 0:
        return _amount
    # no underlying balance, price per share = 0
    if _totalBalance == 0:
        return 0
    # calc shares
    numerator: uint256 = _amount * _totalShares
    shares: uint256 = numerator // _totalBalance

    # rounding
    if _shouldRoundUp and (numerator % _totalBalance != 0):
        shares += 1

    return shares
```

Let's take the following hypothetical example:

- Alice deposit 1000e18 tokens;
- Bob is a malicious user which frontruns Alice with a 1 wei deposit, and a direct transfer of 501e18 tokens to the vault.
- Alice's shares would be: 1001e18 / 501e18 = 1;
- Backrunning Alice's deposit, Bob goes on to withdraw:
- amount: 1501e18 / 2 = 750e18;
- Bob managed to steal a quarter of Alice's assets, the above example is just an example, the attack can be engineered to steal the majority of Alice's deposit. Alice has to receive some shares and thus the whole deposit can't be stolen due to a condition requiring shares != 0.

Recommendations: Make sure that the first deposit is made by the deployer in the same transaction when the vault is deployed to prevent this attack.

Customer's response: Acknowledged

Medium Severity Issues

M-01. Manager can steal the majority of vault assets by re-depositing vault tokens

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: EarnVaultWallet.vy	Status: Fixed	

Description:

The total assets inside of the `EarnVault` are calculated as the current underlying balance of the vault itself, summed with all of its open yield positions' underlying amounts deposited through the Legos, via tracking the yield-bearing tokens and viewing their redeem amounts. This works when the yield-bearing token is directly tied to the underlying assets. However, if those tokens get re-deposited for extra yield, or maliciously through a crafted vault, they leave the calculation and in return - reduce the total assets the system detects, impacting the exchange rate.

A straightforward example is USDC -> aUSDC and then this aUSDC being re-deposited into some staked saUSDC form, which breaks saUSDC's tie to USDC and in turn - does not account towards the supply.

Considering that Euler Earn vaults can be permissionlessly created,, and the current system allows depositing to any Euler vault, even if a vault for "aUSDC" token doesn't exist, it can be created.

```python
# update yield position
    if _asset == VAULT_ASSET:
        assert _vaultAddr == vaultToken # dev: vault token mismatch
        assert staticcall VaultRegistry(_ad.vaultRegistry).checkVaultApprovals(self,
vaultToken) # dev: lego or vault token not approved
        self._updateYieldPosition(vaultToken, _ad.legoId)
```

```
        currentUnderlying += assetAmount
```

The vault only updates positions, if the deposited asset is the underlying asset itself.

Recommendations: Either implement logic to track re-deposited assets, opening the possibility of even more yield, or disable the ability to deposit already registered vault assets to further vaults.

Customer's response: Fixed in f3e9f0e

Fix Review: Fix confirmed

M-02. Managers can swap out to a token which isn't the vault token by using phantom legos

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: EarnVaultWallet.vy LevgVaultWallet.vy	Status: Fixed	

Description:

Similarly to the wallets, the Underscore vaults allow the managers to swap out the assets held, bound by a set of rules. However, we never verify that the provided lego for the swap indeed implements the necessary interface, allowing us to break swap invariants via doing [A -> B -> C], where the A->B swap is the intended swap and B->C uses a 'phantom' lego which simply returns empty values, which go unsanitized.

Python

```
@external
def swapTokens(_instructions: DynArray[wi.SwapInstruction, MAX_SWAP_INSTRUCTIONS])
-> (address, uint256, address, uint256, uint256):
    tokenIn: address = empty(address)
    tokenOut: address = empty(address)
    legoIds: DynArray[uint256, MAX_LEGOS] = []
    tokenIn, tokenOut, legoIds = self._validateAndGetSwapInfo(_instructions)
    ad: VaultActionData = self._canManagerPerformAction(msg.sender, legoIds)

    # key addresses
    usdc: address = USDC
    green: address = GREEN
    savingsGreen: address = SAVINGS_GREEN
    levgData: RipeAsset = self.leverageAsset
    levgVaultHelper: address = self.levgVaultHelper
    origAmountIn: uint256 = _instructions[0].amountIn
    currentBalance: uint256 = staticcall IERC20(tokenIn).balanceOf(self)
```

```
    # important checks!
    assert tokenIn not in [ad.vaultAsset, self.collateralAsset.vaultToken,
levgData.vaultToken, savingsGreen] # dev: invalid swap asset
    if tokenIn == green:
        assert tokenOut == usdc  # dev: GREEN can only go to USDC
    elif tokenIn == usdc and tokenOut != green:
        assert tokenOut == ad.vaultAsset  # dev: must swap into vault asset
        origAmountIn = staticcall
LevgVaultHelper(levgVaultHelper).getSwappableUsdcAmount(
            self,
            origAmountIn,
            currentBalance,
            levgData.vaultToken,
            self.vaultToLegoId[levgData.vaultToken],
            levgData.ripeVaultId,
            usdc,
            green,
            savingsGreen,
            ad.legoBook,
        )

    origAmountIn = min(origAmountIn, currentBalance)
    assert origAmountIn != 0  # dev: no amount to swap

    amountIn: uint256 = origAmountIn
    lastTokenOut: address = empty(address)
    lastTokenOutAmount: uint256 = 0
    maxTxUsdValue: uint256 = 0
    # perform swaps
    for i: wi.SwapInstruction in _instructions:
        if lastTokenOut != empty(address):
            newTokenIn: address = i.tokenPath[0]
            assert lastTokenOut == newTokenIn # dev: path
            amountIn = min(lastTokenOutAmount, staticcall
IERC20(newTokenIn).balanceOf(self))
        thisTxUsdValue: uint256 = 0
        lastTokenOut, lastTokenOutAmount, thisTxUsdValue =
self._performSwapInstruction(amountIn, i, ad)
```

```
        maxTxUsdValue = max(maxTxUsdValue, thisTxUsdValue)

    # verify green <--> usdc swap is fair (check slippage)
    if tokenIn in [green, usdc] and lastTokenOut in [green, usdc]:
        assert staticcall
LevgVaultHelper(levgVaultHelper).performPostSwapValidation(tokenIn, origAmountIn,
lastTokenOut, lastTokenOutAmount, self.usdcSlippageAllowed,
self.greenSlippageAllowed, usdc, green) # dev: bad slippage
    log LevgVaultAction(
        op = 20,
        asset1 = tokenIn,
        asset2 = lastTokenOut,
        amount1 = origAmountIn,
        amount2 = lastTokenOutAmount,
        usdValue = maxTxUsdValue,
        legoId = ad.legoId, # using just the first lego used
        signer = ad.signer,
    )
    return tokenIn, origAmountIn, lastTokenOut, lastTokenOutAmount, maxTxUsdValue
```

There are certain invariants, such as `assert tokenOut == ad.vaultAsset` that need to hold while performing token swaps. The problem is that they can be circumvented either intentionally or unintentionally, by performing a legitimate swap from A to B to C (C is the the lastTokenOut), in which A to B is a swap that takes, place while in the B to C, we never use a lego which can perform swaps, but one that hasn't implemented that functionality, allowing us to break the invariants.

Recommendations: Implement checks to make sure the lastTokenOutAmount is greater than 0.

Customer's response: Fixed in 6dbcbf1

Fix Review: Fix confirmed

M-03. Calculation of maxBorrowAmount in USDC vaults is flawed

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: LevgVaultHelper.vy	Status: Acknowledged	

Description:

In USDC vaults specifically, using the net capital tracked per deposit does not accurately represent the current redeemable amount of underlying tokens, so borrowing is capped harder than it should be. This by itself is ok as a mechanism, since we cannot overborrow in this scenario. However, in the case of a black swan bad market event we could have a negative yield on our positions, thus we are at a loss of tokens. In this case, the reported capital will be larger than the real underlying tokens, thus leading to overborrowing and potential liquidation of the vault, depending on LTV.

Recommendations: Consider keeping the current net capital structure for borrowing on USDC vaults, but for the bottom line, compare it with the vault's current holdings to make sure no overborrowing occurs.

Customer's response: Acknowledged

Low Severity Issues

L-01. Weighted price per share can deviate from the true recent		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: YieldLegoData.vy	Status: Acknowledged	

Description:

The data for every yield lego is tracked separately via price snapshot updates that clear old records based on a current maximum number of snapshots, set by governance. Each index gets written to and once we reach the maximum, the index is reset to 0 and we start overriding the old records, which keeps the TWAP of the shares accurate to recent prices.
The problem here is that this maximum number of snapshots isn't immutable, so it can be increased and reduced, without a follow up clean-up of the records past the cut-off point. This creates a scenario where old records can get re-introduced via decrease->increase of the snapshot cap, leading to a slightly inaccurate TWAP, depending how large the jump between snapshot caps is and if there is staleness enabled (or if it short enough to protect against this scenario):

```python
   def _getWeightedPricePerShare(_vaultToken: address, _lastPricePerShare: uint256) ->
uint256:
    config: ls.SnapShotPriceConfig = self.snapShotPriceConfig
    if config.maxNumSnapshots == 0:
        return 0

    # calculate weighted average price using all valid snapshots
    numerator: uint256 = 0
    denominator: uint256 = 0
    for i: uint256 in range(config.maxNumSnapshots, bound=max_value(uint256)):

        snapShot: ls.SingleSnapShot = self.snapShots[_vaultToken][i]
        if snapShot.pricePerShare == 0 or snapShot.totalSupply == 0 or snapShot.lastUpdate ==
0:
```

```
            continue

        # too stale, skip
        if config.staleTime != 0 and block.timestamp > snapShot.lastUpdate +
config.staleTime:
            continue

        numerator += (snapShot.totalSupply * snapShot.pricePerShare)
        denominator += snapShot.totalSupply

    # weighted price per share
    weightedPricePerShare: uint256 = 0
    if numerator != 0:
        weightedPricePerShare = numerator // denominator
    else:
        weightedPricePerShare = _lastPricePerShare

    return weightedPricePerShare
```

Recommendations: Clear records that are left unused when changing caps on snapshots

Customer's response: Acknowledged

L-02. Clearing a vault token opportunity does not clear its snapshots

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: YieldLegoData.vy	Status: Fixed	

Description:

When an asset opportunity is removed, the code clears `vaultToAsset[_vaultAddr]` and the opportunity indices, but it does not clear `snapShotData[_vaultToken]` or `snapShots[_vaultToken][i]`. If the same vault token is later re-added, historical snapshots remain and will be used by `_getWeightedPricePerShare()` unless they expire via the stale time. Because `staleTime` is allowed to be 0, stale snapshots can permanently contaminate future averages.

Recommendations: On removal, purge snapshot state for the vault token

Customer's response: Fixed in 237b81f

Fix Review: Fix confirmed

L-03. Vault Wallet manager indexes are not properly cleared and could cause confusion

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: EarnVaultWallet.vy LevgVaultWallet.vy	Status: Fixed	

Description:

```python
def removeManager(_manager: address):
    assert self._isSwitchboardAddr(msg.sender) # dev: no perms

    numManagers: uint256 = self.numManagers
    if numManagers == 1:
        return

    targetIndex: uint256 = self.indexOfManager[_manager]
    if targetIndex == 0:
        return

    # update data
    lastIndex: uint256 = numManagers - 1
    self.numManagers = lastIndex
    self.indexOfManager[_manager] = 0

    # get last item, replace the removed item
    if targetIndex != lastIndex:
        lastItem: address = self.managers[lastIndex]
        self.managers[targetIndex] = lastItem
        self.indexOfManager[lastItem] = targetIndex
```

In the above function we remove a manager by taking the lastIndex and the last manager at that index. We:

- set the manager to replace to be the last manager
- we set the index of the last manager to now point to the index of the replaced one
- we reduce the number of managers
- we set the index pointing to the replaced manager to be 0

What we do not do is set the `manager[targetIndex]` to an empty value:

[1 → a, 2 → b 3 → c] - manager map

[a → 1, b → 2 c → 3] - indexOf reverse lookup map

When we want to remove index 1, we get: [a → 0, b → 2, c → 1]

[1 → c, 2 → b, (the length gets reduced, so 3 cannot be touched, unless we add a new manager and override 'c') 3 → c ——this is contained twice now]

Technically we do not have a problem per se, since 'c' will eventually get overridden when adding a new manager. The problem here is that those maps are completely public and can be queried off-chain. The reverse lookup will point 'c' to only 1 index, while the mapping will contain a dirty value that's leftover from the previous removal.

Recommendations: Correctly clean the reverse lookup table of the last swapped element

Customer's response: Fixed in 9f8ffba

Fix Review: Fix confirmed

L-04. If the `maxNumSnapshots` gets reduced, the next snapshot write will be written out of bounds

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: YieldLegoData.vy	Status: Acknowledged	

Description:

When `snapShotPriceConfig.maxNumSnapshots` is reduced, the next call to `_addPriceSnapshot` writes to `snapShots[_vaultToken][data.nextIndex]` using the stale, pre-shrink `nextIndex` before clamping to the new capacity. This writes the new snapshot out of bounds so it will not be considered by the TWAP. Only after the write does `nextIndex` get wrapped to 0, so the first post-shrink snapshot is lost to the reader window.

Python

```
      newSnapshot: ls.SingleSnapShot = self._getLatestSnapshot(_vaultToken, _pricePerShare,
_vaultTokenDecimals, data.lastSnapShot, config)
    data.lastSnapShot = newSnapshot
    self.snapShots[_vaultToken][data.nextIndex] = newSnapshot

    # update index
    data.nextIndex += 1
    if data.nextIndex >= config.maxNumSnapshots:
        data.nextIndex = 0

    # save snap shot data
    self.snapShotData[_vaultToken] = data

    # update cached weighted average price per share
    lastAveragePricePerShare: uint256 = self._getWeightedPricePerShare(_vaultToken,
_pricePerShare)
    self.vaultToAsset[_vaultToken].lastAveragePricePerShare = lastAveragePricePerShare
```

Recommendations: Correctly update the next snapshot index upon reducing the cap on the snapshot number

Customer's response: Acknowledged

Informational Issues

I-01. Collateral/Leverage vault changes can be bricked via 1 wei donations

Description:

The process to change the leverage vault's underlying earn vaults for collateral/leverage requires that the Undy balance of the vault is 0. However, this can easily be bricked by tiny donations:

```python
   # validate old collateral vault token has no balances
    if oldCollData.vaultToken != empty(address):
        assert staticcall LevgVaultHelper(levgVaultHelper).getCollateralBalance(self,
oldCollData.vaultToken, oldCollData.ripeVaultId) == 0 # dev: old vault has ripe balance
        if oldCollData.vaultToken != _vaultToken:
            assert staticcall IERC20(oldCollData.vaultToken).balanceOf(self) == 0 # dev: old
vault has local balance
```

Recommendations: Allow a small buffer to allow the token change, after which invoke a token sweep to remove the dust that might throw off the exchange rate

Customer's response: Acknowledged

I-02. Blacklisted addresses can still redeem vault token for assets

Description:

Even though the Undy token's implementation disallows transfers and mints for blacklisted addresses, it doesn't do so when burning tokens. This effectively allows blacklisted addresses to still redeem shares and interact with the system

```python
@internal
def _burn(_owner: address, _amount: uint256):
    self.balanceOf[_owner] -= _amount
    self.totalSupply -= _amount
    log Transfer(sender=_owner, recipient=empty(address), amount=_amount)
```

Recommendations: If determined to be needed, add the blacklist to burning.

Customer's response: Acknowledged

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.