PUBLIC

Code Assessment

of the yldr.com Smart Contracts

July 28, 2023

Produced for

by

Contents

1 Executive Summary

Dear yldr.com team,

Thank you for trusting us to help yldr.com with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of yldr.com according to Scope to support you in forming an opinion on their security risks.

yldr.com implements a cross-chain yield aggregation system. Users can deposit assets on a master vault on the Ethereum network and then aggregate yield from different protocols in different chains.

The most critical subjects covered in our audit are the bridging mechanism, the interactions with the external protocols, components such as oracles, and the accounting of the system. A critical issue was uncovered, regarding price manipulation by an attacker as well as some high-severity issues. A second critical issue was found in the second iteration which allowed a user to mint more shares than expected by the system. All issues have been addressed.

The general subjects covered are the functional correctness and the liveness of the system, the code complexity, the access control, the documentation, testing, and the gas efficiency. The functional correctness is high. Regarding liveness, we have detected many possible ways which can lead the system to block. A relevant issue has been acknowledged by the development team. However, funds of the protocol are not at risk as the admins are in full control of them. The complexity of the bridging mechanism is high. The documentation was limited especially at the beginning of the review as well as testing. As the system exchanges messages with other chains, interacting with it could be gas-consuming and the gas efficiency is overall improvable. The security, as far as access control is concerned, is high.

In summary, we find that the security of the system is satisfactory but there is room for improvement.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	2
• Code Corrected	2
High-Severity Findings	4
• Code Corrected	4
Medium-Severity Findings	12
• Code Corrected	10
• Acknowledged	2
Low-Severity Findings	2
• Code Corrected	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the yldr.com repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	19 June 2023	ae3ad1f8ddfcf50c36bb6521123d4465a2c030c6	apyflow-dlt V1
2	19 June 2023	d6f3570cf5ca8bfa4e7ecd13b85d1b11f7ab6677	crossledgervault V1
3	19 June 2023	2ad8450cff8fa0669d8c2e924534d177fbe6fadc	posbridgeadapter V1
4	19 June 2023	3d91e09d70f5d542b096f89dfab4837a4bee75b0	wormhole-bridge-adapter V1
5	24 July 2023	c3c3b65e482a3a13ca6f02a6040975814ce48433	apyflow-dlt V2
6	24 July 2023	f6ea6069e3cdd57284c7df4e9893a2f1d0b9c5a1	crossledgervault V2
7	24 July 2023	0bdc9e96d951281ac85b39560e627a37b5f59647	posbridgeadapter V2
8	24 July 2023	8e246146a32e3e14e8c0114edfe0f1ec005f7c64	wormhole-bridge-adapter V2
9	27 July 2023	ef9cdd00c2758f874acac24e2df81ef6f1e3f5fc	crossledgervault V3

For the solidity smart contracts, the compiler version `0.8.19` was chosen.

From `apyflow-dlt` repo the following contracts under `contracts` directory are in scope:

- `protocol-vaults/`:
  - `WrappedERC4626UniswapV3Aave.sol`
  - `WrappedERC4626QuickswapV3Aave.sol`
  - `WrappedERC4626UniswapV3.sol`
  - `WrappedERC4626QuickswapV3.sol`
  - `concentrated-liquidity/`:
    - `BaseConcentratedLiquidityStrategy.sol`
    - `BaseHedgedConcentratedLiquidityStrategy.sol`

        - BaseUniswapV3Strategy.sol
        - BaseQuickswapV3Strategy.sol
    - libraries/:
        - ConcentratedLiquidityLibrary.sol
        - QuickswapV3Library.sol
        - UniswapV3Library.sol
- HarvestableApyFlowVault.sol
- ApyFlowVault.sol
- ApyFlow.sol
- SingleAssetVault.sol
- SuperAdminControl.sol

From crossledgervault repo the following contracts under contracts directory are in scope:

- libraries/:
    - LzMessages.sol
    - Operations.sol
    - SafeAssetConverter.sol
- slippage-providers/ConstantSlippageProvider.sol
- CrossLedgerOracle.sol
- CrossLedgerVault.sol
- LzApp.sol
- MasterCrossLedgerVault.sol
- SlaveCrossLederVault.sol

From posbridgeadapter repo, the following contracts under contracts directory are in scope:

- PosBridgeAdapter.sol
- Worker.sol

From wormhole-bridge-adapter_code repo, the following contracts under contracts directory are in scope:

- Worker.sol
- WormholeBridgeAdapter.sol

2.1.1 Excluded from scope

All the contracts not mentioned explicitly in scope are out of scope. It is important to note that the following contracts are out-of-scope even though they interact with the some contracts in scope:

- AssetConverter.sol
- ChainlinkPriceFeedAggregator.sol

- `PortfolioScoreOracle.sol`

The aformentioned contracts are assumed to work correctly. Furthermore, all the exernal libraries like OpenZeppelin are out of scope. The system relies on external components such as LayerZero and various bridges. These components are assumed to be trusted and work as expected. The system also makes use of oracles which are controlled by its admins. The values returned by these oracles as well as the configuration of the system are assumed to be correct and the admins are expected to never act maliciously. The system interacts with external protocols. These are assumed to work correctly. Finally, the financial strategies implemented were not checked for their validity.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

yldr.com offers a cross-chain yield farming protocol. The system consists of two distinct parts, a bridging mechanism that supports transferring assets across different chains and a protocol integration mechanism. The system consists of slave vaults across different chains, which are all controlled by one master vault on the main chain. Each chain has one deployed root vault contract that controls sub-vaults that integrates with various protocols available on the chain. These root vaults are used as the main access point to yield strategies by master/slave vault contracts.

2.2.1 Bridging Mechanism

The cross-chain communication is implemented between the master vault on the root chain (Ethereum) and the slave vaults on the other chains (currently Polygon and Moonbeam). No direct communication is possible between slave vaults. The communication is implemented with operations that are processed between the two parties. At each point in time, only one operation can be processed and no other operation can begin before the previous one is completed. An operation is initially added to a queue. Any user can call `startNextOperation()` on the Master Vault to start the processing of the next operation. The communication makes use of two components, the respective bridge between the two chains to transfer assets, and Layer Zero to transfer messages between them.

Transferring Assets Between Chains:

Asset transfers begin with a `sendAsset()` call on the respective adapter. This deploys a `Worker` contract in a pre-determined address (`create2`). A worker will be the sender of the assets from one chain. To receive the assets on the other chain, a keeper must call `pullAssets()` of the adapter on the receiver chain. This call will deploy another worker on the receiver chain at the same address as in the sender chain. For the Polygon adapter, when assets are transferred from Ethereum to Polygon, assets become automatically available to the worker's address on Polygon, even if the worker is not yet deployed. When assets are transferred from Polygon to Ethereum, the user must submit some `exitData` when pulling assets. For Moonbeam's wormhole bridge, the assets become available on any bridge only when the respective data is provided.

Exchanging Messages Between Chains:

The message exchanges between bridges are facilitated by the LayerZero (LZ) protocol. For the source chain, a call to the respective LZ entry point is made. The LZ-keepers, through the respective LZ endpoint, will make a call to `lzReceive()` of the respective `LzApp` contract on the destination chain. We discuss the particular types of messages being exchanged in the following sections.

In the following parts, we discuss asset deposit and redemption only to what regards the coordination of the system across chains. We discuss the particular deposits on various protocols in the Wrapped Protocols section.

Depositing Assets:

A user can initiate the assets deposit only on the root chain. Asset deposit will eventually mint shares for the user. A user can deposit any allowed asset. This creates a `DEPOSIT` operation. The asset is swapped to the master vault's main asset. The amounts to be transferred to each chain are determined by the total performance of the portfolio on that chain. For deposits to different chains, a cross-chain transfer is initiated, and an `UPDATE` message is sent through LZ. The actual deposit on any chain is then processed by `processAction()`, which can be called by any user and for any amount up to the expected one. When the full required amount is processed, a `DepositedFromBridge` message is sent back to the root chain through LZ. When all such messages have been received, the shares are mint for the beneficiary.

Redeeming Assets:

Redemptions follow a similar execution plan as deposits. A user specifies an allowed asset to which their shares are going to be converted. The shares of the user are first transferred to the master vault and burned once the operation is completed. A `Redeem` message is sent to all the chains through LZ. The action is processed on each chain by a separate call to `processAction()` which makes a redeem call to the root vault on the current chain. Fees are paid during this step. When the redemption from all the chains is completed, the assets collected are transferred to the beneficiary.

Rebalancing:

Rebalancing can only be initiated by the `REBALANCER_ROLE`. Rebalancing corresponds to a redemption on one chain which sends the assets to the master vault and a deposit to a different chain. A rebalance operation is not added to the queue. It is expected instead to execute immediately.

2.2.2 RootVaults and Single Asset Vaults

Master and Slave vaults deposit assets to the root vault of each chain. Each such root vault consists of one `ApyFlow` vault, with multiple so-called single asset vaults (SAV) as underlying. An SAV also consists in turn, of many vaults which all use the same underlying asset e.g., USDC.

Root Vaults:

The interface of the root vault includes but is not limited to the following functions:

- `deposit()`: any user can deposit in the underlying asset of the vault. The assets are split among the SAVs based on the portfolio score of each SAV and shares are minted for the specified receiver.
- `redeem()`: any user can burn part of their shares in exchange for some assets which are sent to a specified receiver. The amount to be redeemed from each SAV is determined by the portion of shares against the total supply of shares that are burnt. A fee is paid during redemption, which is sent to the treasury.
- `rebalance()`: As with master and slave vaults, a rebalance constitutes redemptions from an SAV and a deposit to another SAV. Any user can trigger the rebalance as long as a valid amount of shares to be reallocated is defined. This amount is determined by the score deviation of each SAV. The score of an SAV corresponds to the portion of the total value of the root vault held by this specific SAV.
- `add/removeVault()`: the owner of the vault, i.e., a master or a slave vault repsectively can add and remove SAVs.

Single Asset Vaults

The interface of single asset vaults is pretty similar to the root vaults'. The main difference is that SAVs interact with Wrapped Vaults instead and the score deviation is concerned with the vault of the Wrapped Vaults. Moreover, during redemption, no fee is withheld at this level.

2.2.3 WrappedVaults

yldr.com integrates with multiple protocols. The interactions with the external protocols are implemented with different Wrapped Vaults which hold the shares of the respective protocol. Wrapped vaults using the same underlying are organized in Single Asset Vaults (SAV). Different SAVs are organized under Root Vault. There's only one root vault per chain. In that sense, the vaults are organized in a tree-like structure. It is important to note that any user can directly interact with the vaults at any level of the tree i.e., root vault, single asset vault, or wrapped vault. In this review, we focus solely on the Wrapped vaults. The owner of the vault can execute any action on behalf of the vault (see Trust Model).

UniswapV3

A user can participate in a UniswapV3 liquidity position and earn fees.

- `deposit()`: it adds liquidity to the lp-position of the vault if such a position exists. It creates a new position otherwise around the current price.
- `redeem()`: it redeems the part of the position that corresponds to the shares a user wants to burn. The redeemed amount includes the fees which are collected for that position.
- `readdLiquidity()`: it redeems the whole position and redeposits the total value.

UniswapV3Aave

This strategy implements a hedged LP Uniswap position. The user deposits only one asset of the Uniswap pair which is also used as collateral on Aave to borrow the other asset of the pair.

- `deposit()`: If the position doesn't exist, it simply calculates the amount of the asset that needs to be provided as collateral on Aave, it borrows the amount of the missing token and deposits both tokens on UniswapV3 to mint a position. In case a position already exists, the user creates extra debt on Aave proportional to their contribution. If the amount of the missing asset e.g., ETH is more than the needed one, a part of it is converted to the collateral asset. If more ETH is needed part of the collateral asset i.e., USDC is converted to ETH.
- `redeem()`: A user burns part of their shares of that particular vault in exchange for some assets of the vault. Initially, part of the liquidity in the pool is redeemed and appropriately converted to the borrowed asset so that a part of the respective part of the debt is returned to Aave. The remaining amount is all converted to the collateral asset and sent to the owner of the shares.
- `readdLiquidity()`: This is equivalent to `readdLiquidity()` on UniswapV3 strategy. However, Aave doesn't allow closing and opening a debt position on the same block. This means, that a series of actions can take place to emulate `readdLiquidity()` on Uniswap.

QuickswapV3Aave

This strategy is similar to UniswapV3Aave but it aims Quickswap instead. The only difference is that Quickswap implements some farming functionality that needs to be taken into consideration depositing and withdrawing.

2.2.4 Roles and Trust Model

In the current section, we're defining the trust model of the system.

Main Roles

- **Super Admin Control Owner**

  Access Control Type: Openzeppelin's `Ownable`, ownership can only be transferred by the current owner.

  Role: Emergency actions

  Trust Assumption: Fully trusted

Abilities: The role is in control of the funds in the vault. They can transfer any token anywhere and call any smart contract with any data and value on behalf of the vault.

- **Portfolio Oracle Owner**

  Access Control Type: Openzeppelin's `Ownable`, ownership can only be transferred by the current owner.

  Role: Setting the appropriate score for all chains/vaults.

  Trust Assumption: Fully trusted.

  Abilities: The role can arbitrarily set the portfolio scores. Rebalance actions highly depend on these scores, repeated rebalancing can cause some loss to the protocol.

- **Price Feed Owner**

  Access Control Type: Openzeppelin's `Ownable`, ownership can only be transferred by the current owner.

  Role: Setting the appropriate external oracle source for every token.

  Trust Assumption: Fully trusted.

  Abilities: The role can arbitrarily set the price sources. Most actions on the vault depend on these prices. For example, a malicious price feed owner could devaluate some vaults and mint more shares than they should.

- **Converters Owner**

  Access Control Type: Openzeppelin's `Ownable`, ownership can only be transferred by the current owner.

  Role: Specifying up-to-date swap routes data.

  Trust Assumption: Fully trusted

  Abilities: The converter is used across most vaults and their actions, it also controls acceptable slippage for swaps. Big slippage can lead to a denial of service. Moreover, wrong amounts can end up being deposited on external protocols.

- **CrossLedgerVault Admin Role**

  Access Control Type: Openzeppelin's `AccessControl`, `ADMIN_ROLE` is self-controled.

  Role: Adding, removing, and configuring slave chains, as well as their bridge adapter.

  Trust Assumption: Fully trusted

  Abilities: Bridge adapters have full allowance over the main asset. Trusted remotes can send any LZ messages across vaults. Whitelisting a malicious remote can allow a malicious admin to create and send arbitrary messages which can break the system.

- **MasterCrossLedgerVault Admin Role**

  Access Control Type: Openzeppelin's `AccessControl`, `ADMIN_ROLE` is self-controled.

  Role: Adding and removing tokens, setting the `minimumOperationValue` as well as the minimum slippage provider.

  Trust Assumption: Fully trusted

  Abilities: Inherits all CrossLedgerVault Admin's abilities.

- **Rebalancer Role**

  Access Control Type: Openzeppelin's `AccessControl`, `REBALANCER_ROLE` is administrated by `ADMIN_ROLE`.

  Role: The role can execute rebalances.

Trust Assumption: Fully trusted

Abilities: It can arbitrarily rebalance between different chains. Thus, it can influence the overall performance of the portfolio, DoS the protocol, or cause some arbitrary slippage loss.

- **Users**

  Role: End users of the protocol.

  Trust Assumption: Not trusted

  Abilities: Interacting with external/public functions of the protocol.

Out of scope trusted actors

- **Price oracles:** Prices should reflect the real market price of assets, more specifically, they shouldn't be manipulatable.
- **External market makers both for asset swaps and LP provision**
- **Aave**

General assumptions about these external actors for the protocol to work properly are:

- They aren't in a paused state, they work as expected.
- They do not allow externally controlled reentrancy possibilities. (Only the protocol user decides who can potentially recover the control flow)

2.2.5 Version 2

The changes in the second version of the report include but are not limited to the following:

- For the master cross-ledger vault, the access control is no longer implemented with Openzeppelin's respective library. The balancer role is defined as a storage variable and the most critical operations are allowed for the owner of the contract.
- The admins of the master cross-ledger vault are in full control of the assets as the contracts inherit the super-admin capabilities.
- For the wrapped vaults, deposits, redemptions and rebalances require that there's no deviation between the price reported by the oracles and the price offered by the pools.
- The owners can now skip queued operations in the Master cross-ledger vault as long as there's no operation being processed.
- The owners of the UniswapV3Aave strategy can set the loan to value (LTV) parameter.

2.2.6 Version 3

The most important change in this version is that the operations in the master cross-ledger vault can be skipped at any time by the owners.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2

- Rounding Error Leads to Blocked Queue Acknowledged
- Inflating Shares Price Acknowledged

Low-Severity Findings	0

5.1 Rounding Error Leads to Blocked Queue

Design Medium Version 2 Acknowledged

CS-APYF-001

Rounding errors can block the processing of an operation eventually blocking the whole system. There are multiple instances of this issue.

- Redeeming a small amount through the master vault could round down to zero one of the shares computation per chain. However, a zero amount shares redemption on `ApyFlow.redeem()` reverts because a computation uses the number of shares as a denominator. The following computation fails in `ApyFlow._redeem()`:

```
uint256 processedPricePerToken = (valueInAsset * (10 ** decimals())) / shares;
```

- In `BaseConcentratedLiquidityStrategy._redeem()`, a rounding error in the calculation of the liquidity to be removed can lead the system to block as UniswapV3/QuickswapV3 does not allow removing 0 liquidity:

```
    uint128 liquidity = uint128(
    (_getPositionData().liquidity * shares) / totalSupply()
);
```

- In `BaseConcentratedLiquidityStrategy._deposit()`, a rounding error in the calculation of the liquidity to be added can lead the system to block as UniswapV3/QuickswapV3 does not allow adding 0 liquidity.
- The same issues as above appears in `BaseHedgedConcentratedLiquidityStrategy`.

Acknowledged:

yldr.com replied:

> There always will be ways of making deposit/redeem revert at some step of processing and block the queue. Such attacks are non-profitable and relatively expensive for an attacker and considered non-likely. Even if attacks will happen, we have admin functionality which allows skipping of such malicious operations or funds recovery in case of vault becoming fully non-functionable.

5.2 Inflating Shares Price

Security Medium Version 1 Acknowledged

CS-APYF-002

The exchange rate for all `ApyFlowVault` contracts depends on some balances of the assets in the smart contract. Consequently, any asset donation will be taken into account and increase the exchange rate of the vault, and the value of the shares.

A vault with a very small or equal to zero `totalSupply`, is therefore vulnerable to a price inflation attack.

A malicious user could first mint a small number of shares and then send a great amount of assets to the smart contract.

Any subsequent deposit would make the computation `assets.mulDiv(totalSupply_, totalAssets_)` round to zero if the deposit amount is smaller than the exchange rate, leading to assets being transferred to the contract but no shares minted in exchange.

Even if the amount is greater than the exchange rate, the rest of the division would not be accounted for because of the rounding error, leading to a partial donation of the funds to the current vault shares holders.

Client acknowledged and replied:

> Our standard procedure is to deposit a small amount of funds (5-100 USD) into each newly created vault to test if it's functionable and to avoid an inflation attack.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	2

- Minting More Shares Code Corrected
- Price Manipulation Code Corrected

High-Severity Findings	4

- A Deposit Could Be Stuck Because of Slippage Code Corrected
- Fees Are Not Properly Harvested Code Corrected
- Missing Calculation Code Corrected
- Aave.ltv Cannot Be Updated Code Corrected

Medium-Severity Findings	10

- Underflow in Deposit Blocks the System Code Corrected
- Zero Redemption Blocks the System Code Corrected
- Dust in Deposits Code Corrected
- Idling Assets Unused in readdLiquidity() Code Corrected
- Leftovers Are Not Handled Code Corrected
- Queue Processing Is Slow Code Corrected
- Redemption of Small Amounts Is Impossible Code Corrected
- SlaveCrossLedgerVault With a Zero Portfolio Score Code Corrected
- Unused Function Code Corrected
- minimumOperationValue Could Block Redeems Code Corrected

Low-Severity Findings	2

- Discrepancy Between Computed and Actual Price per Token Code Corrected
- Queue Could Be Stuck Because of Small-Amount Swaps Code Corrected

6.1 Minting More Shares

Security Critical Version 2 Code Corrected

CS-APYF-027

The `_startDeposit()` function in the `MasterCrossLedgerVault` iterates over each chain and deposits an amount proportional to their associated portfolio score returned by the oracle. In the case where the amount destined to this chain rounds down to zero, the chain is ignored in `_deposit()` and the loop continues and no message is sent to and received from the chain. This can allow a malicious user to mint more shares than the value they deposited.

The amount of shares minted is computed in the `MasterCrossLedgerVault` as follows:

```
uint256 totalAssetsBefore = totalAssetsBeforeDeposit[opId];
uint256 totalAssetsAfter = totalAssetsAfterDeposit[opId];
uint256 deposited = totalAssetsAfter - totalAssetsBefore;
uint256 shares;
if (totalSupply() == 0) {
    shares = totalAssetsAfter;
} else {
    shares = (deposited * totalSupply()) / totalAssetsBefore;
}
```

For a fair computation of the shares, `totalAssetsBefore` should denote the **full** value of all assets across all chains. `totalAssetsBeforeDeposit[opId]` is increased after a message is received from a slave chain that contains the total value of the assets held on the chain before the deposit of the user's assets. As no deposit is performed on the slave chain no such message is received.

The next important aspect to understand is the `deposited` amount does not depend on the actual amount deposited by the user when the initiate the deposit. If the attacker donates to a pool used by an asset converter in the same transaction, the asset converter can receive a higher amount of the output token than the current price. This allows the attacker to take advantage of the rounding down errors caused by the small amount originally deposited while they eventually deposit a big amount to the system.

Since the shares minted depend on the ratio `deposited/totalAssetsBefore`, the attacker is able to mint a big amount of shares as they increased `deposited` (by manipulating the pool) while keeping `totalAssetsBefore` low (by skipping chains).

The attacker can now redeem their shares. Redemption works a bit differently than depositing: It just iterates over all chains without taking into account the portfolio score but instead provides them with a relative proportion (shares & totalSupply), so that each `SlaveCrossLedgerVault` can use this proportion to compute how many assets to redeem.

However, for `totalAssetsBefore` to be updated by a slave vault, a cross-chain deposit must have occurred, which isn't the case when the argument is zero in `_deposit()`.

To sum up the attack, the attacker would:

1. Call `MasterCrossLedgerVault.deposit()` with a small amount. This makes every amount round down to zero except for one chain which would have a very small amount deposited.
2. Increase this deposited amount by for example manipulating a liquidity pool.
3. Steal funds by directly redeeming the inflated number of shares minted.

Code corrected:

A new type of message has been introduced: `ZERO_DEPOSIT` to handle the case where the amount rounds down to zero. This allows chains to communicate their total assets even though no funds were deposited during the operation.

6.2 Price Manipulation

Design | Critical | Version 1 | Code Corrected

CS-APYF-026

Concentrated liquidity strategies must be able to compute the price and value of their liquidity position for price range adaptation and accounting purposes. However, current price computations only use manipulable pool data, which can consequently not be relied on to provide real market price and value.

Such manipulation leads to critical issues:

6.2.1 Manipulate the price range rebalance process:

The concentrated liquidity strategies should automatically adapt their price range when the current pool price is above or under the currently set range. This process happens in the function `readdLiquidity()`, which heavily depends on the current pool tick. The pool tick is heavily manipulable and could be used by an attacker to make the strategy provide liquidity at a manipulated price.

```
function readdLiquidity() public virtual {
    ...
    (int24 currentTick, ) = _getPoolData();
    int24 tickLowerToRebalance = data.tickLower + ticksUntilRebalance;
    int24 tickUpperToRebalance = data.tickUpper - ticksUntilRebalance;
    bool isInRebalanceRange = (tickLowerToRebalance > currentTick) || (tickUpperToRebalance < currentTick);
    ...
    require(
        isInRebalanceRange ||
            (pricePerTokenAfter >= (lastPricePerToken * 1001) / 1000)
    );
```

An example attack flow on a `USDC-WETH` pool strategy:

1. Borrow a lot of `ETH` in a flash loan
2. Sell all `ETH` in the strategy's pool: Price and tick are now manipulated, price of `ETH` related to `USDC` is much lower than the real market price
3. Call `readdLiquidity()` on the strategy: The current tick is now in the rebalance range, and the function executes and moves liquidity around the current tick
4. Buy back *ETH* in the pool with every `USDC` received from step 2: Liquidity moved in step 3 is now at a very advantageous price for the attacker, who makes a profit out of the strategy's funds.
5. Repeat

Note that this attack depends on the strategy's liquidity size, as well as the pool's size and the fees.

6.2.2 Manipulate the strategy's exchange rate:

The shares exchange rate depends on the `totalAssets()` function that computes the total value of the strategy's funds.

```
function totalAssets() public view override returns (uint256) {
    ...
    (uint256 amount0, uint256 amount1) = LiquidityAmounts
        .getAmountsForLiquidity(
            sqrtPriceX96,
            sqrtPriceAX96,
            sqrtPriceBX96,
            _getPositionData().liquidity
        );
    uint256 valueInUSD;
    valueInUSD += pricesOracle.convertToUSD(token0(), amount0);
    valueInUSD += pricesOracle.convertToUSD(token1(), amount1);
```

Here, `amount0` and *amount1* will depend on `sqrtPriceX96`, which is the current price of the pool. These amounts will then be converted to their real current market value in USD (which may not be close to the pool price).

Note that at any point in time, some assets might be idling in the smart contract.

This issue could lead to multiple consequences:

- An attacker could potentially lower the price of the liquidity position and deposit assets that will be overvalued proportionally to the liquidity position, resulting in some extra shares being minted to the attacker.
- Inflating the value of the strategy shares if they are used in an external protocol (for example as collateral).
- Front-running a user deposit/redeem could become profitable.

6.2.3 Forced slippage:

`readdLiquidity()` is allowed to execute only if either the price of the pool is out of the current liquidity range or if the price per token after execution has increased. However, being able to manipulate the price of the liquidity pool makes it possible to execute the function on request.

An attacker could use this ability to force the pool to lose value in fees and slippage because of swaps happening during execution.

Note that this list of potential consequences isn't exhaustive as most function that rely on `totalAssets()` are vulnerable.

Code corrected:

The modifier `BaseConcentratedLiquidityStrategy.checkDeviation()` has been implemented. The modifier checks whether the price of a pool deviates more than the `allowedPoolOracleDeviation` from the price reported by an external oracle. If this condition is not true the execution reverts. This means that users are unable to redeem and withdraw their assets during this period of time. Moreover, deposits iniated from the Master vault will be blocked for this period.

6.3 A Deposit Could Be Stuck Because of Slippage

Design | High | Version 1 | Code Corrected

CS-APYF-014

In `MasterCrossLedgerVault`, processing a deposit operation in the queue starts by swapping the deposited asset with the main asset of the vault. To swap in between assets, the vault uses an `AssetConverter`, which checks that the slippage does not exceed a fixed value.

```
function _startDeposit(
    ...
) internal {
    uint256 mainAssetValue = assetConverter.safeSwap(
        params.asset,
        address(mainAsset),
        params.value
    );
```

```
    ...
}
```

However, in the case of a big deposit where the deposited asset is not the same as the main asset, swapping could be impossible without causing a greater slippage than the maximum expected one.

In this case, the vault operations would be stuck and could not continue to process normally until the slippage is changed in the asset converter.

Code corrected:

The swap is now executed before the operation is queued.

6.4 Fees Are Not Properly Harvested

Correctness | High | Version 1 | Code Corrected

CS-APYF-018

The `ApyFlow` smart contract has a `feeInPpm` used to compute a fee applied over all vault profit and that will be sent to the `feeTreasury`. These fees can be harvested by calling `recomputePricePerTokenAndHarvestFee()`, which computes the revenue across all vaults, and applies the fee by minting a proportional share amount.

Let's take a look at how it works internally:

```
function recomputePricePerTokenAndHarvestFee() public {
    uint256 _totalAssets = totalAssets();
    uint256 _totalSupply = totalSupply();

    uint256 newPricePerToken = pricePerToken();
    if (newPricePerToken > lastPricePerToken) {
        ... // Fee shares computation
        _mint(feeTreasury, shares);

        lastPricePerToken = newPricePerToken;
        emit FeeHarvested(fee, block.timestamp);
    }
}
```

`lastPricePerToken` is used to know whether or not the price per token increased since the last time fees were minted. Note that after minting the fees, the last price per token is updated to `newPricePerToken`, which was computed before fees were minted.

However, because of the new shares, the `pricePerToken` just decreased which isn't accounted for when assigning `lastPricePerToken`.

`lastPricePerToken` is now greater than `pricePerToken`, meaning that no fees will be applied until the price per token reaches again the `lastPricePerToken`.

Code corrected:

`lastPricePerToken` is now updated with the most recent value of `pricePerToken()` which includes the harvested fees.

6.5 Missing Calculation

Correctness High Version 1 Code Corrected

CS-APYF-023

`BaseHedgedConcentratedLiquidityStrategy._readdLiquidity()` calculates the amount to be withdrawn from Aave which is stored in `amountToWithdraw`. However, this variable is only declared but not assigned, which initializes it to zero.

Code corrected:

The missing computation was added.

6.6 `Aave.ltv` Cannot Be Updated

Design High Version 1 Code Corrected

CS-APYF-025

When `HedgedBaseConcentratedLiquidityStrategy` is initialized, the `AaveLibrary.Data` is set which stores the aimed `ltv` of the Aave position. `ltv` can never be updated. This can be problematic. The `ltv` on Aave can be different or change during the lifetime of the strategy. This means that the `tvl` on the strategy can be greater than the actual `tvl` on Aave. When the strategy tries to borrow from Aave, the transaction will revert as the strategy will request more to borrow. This limitation of the strategy can prevent deposits and redemptions in the whole system as for a system-wide operation to succeed all the operation in all vaults should succeed. Another important point is that in AaveV3, if `tvl` is 0 for a specific collateral this collateral cannot be withdrawn.

Code corrected:

The function `HedgedBaseConcentratedLiquidityStrategy.updateAaveLTV()` has been added. The owner of the smart contract can now set a new `ltv` value.

6.7 Underflow in Deposit Blocks the System

Design Medium Version 2 Code Corrected

CS-APYF-017

A user can submit a 0 deposit which can be added to the queue. This will eventually execute `_depositLocal` for `assets == 0` which eventually calls `_decreaseOpIdToActionsCount` which decreases `opIdToActionsCount` which is 0. This means that the operation will revert. Note that the owners at this point cannot call `setNewNextOperation` as the queue is busy.

Code corrected:

The `opIdToActionsCount` is now incremented before the rest of the deposit logic, which fixes the underflow case.

6.8 Zero Redemption Blocks the System

Design | Medium | Version 2 | Code Corrected

CS-APYF-010

A user can create a request to redeem 0 shares from `MasterCrossLedgerVault` which will be successfully added to the queue. Such a request can be fully processed up until `_completeRedeem` which will calculate the `pricePerToken`. However, this calculation will revert blocking the completion of the operation since 0 shares are in the denominator. Note that the owners at this point cannot call `setNewNextOperation` as the queue is busy.

Please note that the `pricePerToken` cannot be removed as the following issue will be enabled:

- Assume a system with one slave chain with a score higher than the master chain.
- A user deposits a very small amount such that the deposited amount for the master chain is zero and non-zero for the slave chain.
- `opIdToActionsCount[opId]` is increased once for the operation on the slave chain
- `depositLocal()` is then executed and since assets is 0 for the local chain `_finalizeCurrentAction()` is called.
- Then, `_decreaseOpIdToActionsCount()` is called which sets `opIdToActionsCount[opId]` to 0 which **successfully** calls `_completeOperation()`.
- `operationsQueue.currentOperation` is set to 0 which allows the next operation to be executed.

Code corrected:

Zero shares redemption is not allowed anymore.

6.9 Dust in Deposits

Design | Medium | Version 1 | Code Corrected

CS-APYF-016

In `MasterCrossLedgerVault` and `ApyFlow`, deposited assets are split based on a portfolio score. However, it could be that there are some dust amounts left in these smart contracts due to rounding errors.

`MasterCrossLedgerVault`'s dust would be unrecoverable for the user. `ApyFlow`'s dust is accounted as a donation to the current shareholders.

The last deposited amount (local deposit) could be calculated as the remaining available amount and thus dust would be avoided.

Code corrected:

`MasterCrossLedgerVault` now correctly handles deposit dust by minting the appropriate shares amount and a new `dustAmount` variable has been introduced to keep track of the current dust and split it accordingly on redeems.

`ApyFlow` now also correctly takes dust into account.

6.10 Idling Assets Unused in `readdLiquidity()`

Design | Medium | Version 1 | Code Corrected

CS-APYF-024

Both in `BaseConcentratedLiquidityStratey` and `BaseHedgedConcentratedLiquidityStrategy` the idle assets are not invested when `readdLiquidity` is called even though they are accounted in `totalAssets()`. This results in the strategy giving up on some yield.

- In `BaseConcentratedLiquidityStrategy._readdLiquidity()`, only assets accounted for in `_redeem()` will be deposited back. However, some idling assets might be present in the strategy and will stay idling even after the exection of `readdLiquidity()`.
- `BaseHedgedConcentratedLiquidityStrategy._readdLiquidity()` redeems all concentrated liquidity and adapts its debt and collateral to approach the target ltv. To compute the amount of assets that are available in the vaults, the function calls `_totalAssets()`, which computes the total value in the Aave position, and then accounts for the token balances of the concentrated liquidity pool. However, note that it is possible that the main vault asset isn't equal to either token of the liquidity pool. In this case, idling assets present in the vault are not accounted for, which will lead to a smaller Aave position than expected after execution.

Code corrected:

During `_readdLiquidity()` the total balance of asset held by the contract (not just the amount redeemed) is used in the new deposit.

6.11 Leftovers Are Not Handled

Correctness | Medium | Version 1 | Code Corrected

CS-APYF-013

The execution of `BaseHedgedConcentratedLiquidityStrategy._readdLiquidity()` can leave the contract with some extra `token0` and `token1` idling in the smart contract. This happens because swaps of tokens can return a greater amount of assets than what is needed.

The assets which are not of type `asset` will not be accounted for in `totalAssets()`. This could potentially reduce the value of the vault's shares until the next `_harvest()` function execution, which will swap them back to `asset`.

Code corrected:

The leftovers are converted to the underlying asset at the end of the execution.

6.12 Queue Processing Is Slow

Design | Medium | Version 1 | Code Corrected

CS-APYF-021

`MasterCrossLedgerVault`'s operation queue is very slow to process, as each operation must pass messages across multiple chains.

Waiting time for an operation to execute could be days or even weeks depending on usage.

Even with a reasonably big `minimumOperationValue`, a wealthy malicious user could also easily increase processing time by days, or even weeks.

Code corrected:

`MasterCrossLedgerVault.setNewNextOperation()` was introduced. The owner of the contract can arbitrarily set the next operation.

6.13 Redemption of Small Amounts Is Impossible

Design | Medium | Version 1 | Code Corrected

CS-APYF-011

In `MasterCrossLedgerVault`, a deposit or a redemption is accepted only if the amount is greater than the `minimumOperationValue`:

```
require(
    shares / 10 ** decimals() >= minimumOperationValue,
    "Redeem value is lower than minimum"
);
```

However, users might be willing to partially redeem their position. In some cases it could leave them with an amount of shares that is smaller than the `minimumOperationValue`, making it impossible to redeem the rest of the funds unless they deposit again to reach the minimum operation value.

Code corrected:

The `minimumOperationValue` was removed.

6.14 SlaveCrossLedgerVault With a Zero Portfolio Score

Design | Medium | Version 1 | Code Corrected

CS-APYF-029

The `_startDeposit()` function in the MasterCrossLedgerVault iterates over each chain and deposits an amount proportional to their associated portfolio score returned by the oracle. In the case where either a chain has a score of zero or the amount destinated to this chain rounds down to zero, the loop continues to iterate and ignores this chain:

```
for (uint256 i = 0; i < chains.length(); i++) {
    uint256 chainId = chains.at(i);
    uint256 score = oracle.portfolioScore(chainId);
    uint256 amountToSend = (mainAssetValue * score) / totalScore;
    **if (amountToSend == 0) continue;**
    _deposit(opId, chainId, amountToSend, params.slippage);
}
```

Let's also recall how the amount of shares to be minted is computed in the `MasterCrossLedgerVault`:

```
uint256 totalAssetsBefore = totalAssetsBeforeDeposit[opId];
uint256 totalAssetsAfter = totalAssetsAfterDeposit[opId];
uint256 deposited = totalAssetsAfter - totalAssetsBefore;
uint256 shares;
if (totalSupply() == 0) {
    shares = totalAssetsAfter;
} else {
    shares = (deposited * totalSupply()) / totalAssetsBefore;
}
```

For a fair computation of the shares, `totalAssetsBefore` should denote the full value of all assets across all chains. This is important because redeeming works a bit differently than depositing. It just iterates over all chains without taking into account the portfolio score but instead provides them with a relative proportion (shares & totalSupply), so that each `SlaveCrossLedgerVault` can use this proportion to compute how many assets to redeem.

However, for `totalAssetsBefore` to be updated by a slave vault, a cross-chain deposit must have occurred, which isn't the case when `amountToSend == 0`.

If a portfolio score of a chain has been set to zero, but some assets are still waiting to be redeemed on the slave vault, then funds would be incorrectly distributed to new master vault depositors. A new depositor would receive shares relative to the total assets of all chains except the ones with zero scores, but redeeming these shares would still withdraw funds on the zero-score chain.

Code corrected:

Version 2:

yldr.com replied:

> We have restricted setting 0 score for chains. All chains should be removed instead of zeroing their scoring

`CrossLedgerOracle.updateDataBatch()` is implemented to revert if a score is set to 0 for any chain of a slave vault.

Version 3:

Setting a score of 0 for a chain is now allowed because zero amounts are now properly handled during operations execution.

6.15 Unused Function

Correctness | Medium | Version 1 | Code Corrected

CS-APYF-015

`WormholeBridgeAdapter.adjustAmount()` is never used.

Code corrected:

`adjustAmount()` is now used to round down the last decimals of the amount sent over the wormhole bridge. This is done as the wormhole bridge performs a similar operation.

6.16 `minimumOperationValue` Could Block Redeems

Design | Medium | Version 1 | Code Corrected

CS-APYF-022

The `minimumOperationValue` variable exists as a lower limit of how many assets one can deposit or redeem on `MasterCrossLedgerVault`. It is denominated in US dollars, and can be compared against deposited assets as long as these are stablecoins pegged to the USD and decimals are adapted.

On redeems, however, `minimumOperationValue` is compared against a master vault shares amount, which has a varying price:

```
require(
    shares / 10 ** decimals() >= minimumOperationValue,
    "Redeem value is lower than minimum"
);
```

Assuming that shares accrue in value over time, the minimum value that a user must have deposited to be able to redeem will also increase with time. Note that shares might also decrease in value over time.

A user could deposit but isn't able to redeem afterward because one share is more valuable than one token of the deposited asset. This would lock users' funds until they deposit again to reach the minimum value.

Code corrected:

The `minimumOperationValue` was removed.

6.17 Discrepancy Between Computed and Actual Price per Token

Correctness | Low | Version 1 | Code Corrected

CS-APYF-019

In `CrossLedgerVault.processAction()`, the user-specified slippage is checked against the received assets to make sure it is acceptable for this action. When redeeming, `expectedAssets` is computed with `feeInclusivePricePerToken()` of the underlying root vault, which for now doesn't contain the harvested rewards of the underlying vaults.

However, it might be the case that on redemptions, rewards are harvested, which changes the `pricePerToken` before the redemption happens. Consequently, the `feeInclusivePricePertoken()` will return a smaller price per token than it should have, which would allow a bigger slippage than desired.

Code corrected:

The expected assets are now computed after the redemption so that harvested rewards are accounted.

6.18 Queue Could Be Stuck Because of Small-Amount Swaps

Design Low Version 1 Code Corrected

CS-APYF-012

In `CrossLedgerVault.processAction()`, the amount to deposit or redeem could be small, and lead to some rounding errors depending on the pool used to swap assets in vaults. Depending on the rounding error, the slippage might happen to always be greater than expected, which would block the queue.

Code corrected:

The amount received is always increased by 100 wei so that low amounts cannot make the slippage check fail.

Note that this means bypassing the user-defined accepted slippage for small deposits.

6.19 Redundant Storage

Informational Version 1 Code Corrected

CS-APYF-028

Redundant storage is used in some places. More specifically:

- `CrossLedgerVault.addChain()` sets `lzChainIdToChainId`. However, mapping is never used.
- `CrossLedgerVault.chains` variable is only of use in the `MasterCrossLedgerVault`, not in the slave vaults.
- `WormholeBridgeAdapter.isRootChain` variable is never used.

Code corrected:

The redundancies have been removed.

6.20 Variables Could Be Immutable

Informational Version 1 Code Corrected

CS-APYF-020

The following variables are only set on construction and never written to afterwards.

- `WormholeBridgeAdapter.workerImplementation`
- `PosBridgeAdapter.isRootChain`
- `PosBridgeAdapter.asset`
- `PosBridgeAdapter.workerImplementation`
- `PosBridgeAdapter.dstChainId`
- `PosBridgeAdapter.crossLedgerVault`

- `PosBridgeAdapter.rootChainManager`

Setting these to `immutable` will insert them into the bytecode at compilation time, leading to cheaper reads compared to storage.

Code corrected:

The variables are now immutable.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Code Consistency

Informational **Version 1**

CS-APYF-003

Some code areas lack consistency when dealing with similar logic situations. The `BaseHedgedConcentratedLiquidityStrategy` swaps assets from one token to another in order to supply an appropriate amount to Aave and eventually to the LP position. In multiple cases, extra care is taken so that no underflows take place. For example in `_redeem()` the amount to swap is bounded by the available `collateralAmount`:

```
amountToSwap = Math.min(amountToSwap, collateralAmount);
```

While similar logic is implemented in most cases, there are cases where it's not implemented:

- In `_readdLiquidity()` in the case where `currentDebt > neededDebt` and `tokenToBorrowBalacne < amountToRepay`, `collateralBalance` is assumed to be greater than `amountToSwap`.
- In `_readdLiquidity()` in the case where `currentCollateral < neededCollateral` and `collateralBalance < amountToSupply`, `tokenToBorrowBalance` is assumed to be greater than `amountToSwap`.

Enforcing code consistency when handling specific similar cases helps to secure the code flow and to make it more understandable. Adding internal functions for specific tasks could help to properly split the logic.

7.2 Missing Natspec

Informational **Version 1**

CS-APYF-004

Most functions are missing proper documentation and description.

Natspec help the end users to interact with smart contracts as they produce messages that can be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).

7.3 Missing Sanity Checks

Informational **Version 1**

CS-APYF-005

- During a rebalance operation in `MasterCrossLedgerVault`, the `REBALANCE_ROLE` specifies the percentage of shares to be moved from one chain to another. This happens assuming that the total supply of shares is 1000. However, no sanity check guarantees that the `shareToRebalance` is

less than 1000. Note that the accepted slippage and both the source and destination chain id are also specified but not sanity checked, which in the worst case (slippage input error) could lead to some loss of assets.

- `MasterCrossLedgerVault.setNewNextOperation()` allows the owner to set the next operation id, however, this value isn't sanity checked and could point to an already processed operation.
- The beneficiery for deposits and redemptions is not checked to be non-zero.
- In `ApyFlow.setNewFeeDestination()`, the `newFeeDestination` is not sanitized.

7.4 No Events for Important State Changing Operations

Informational **Version 1**

CS-APYF-006

Some examples are:

- `MasterCrossLedgerVault.addToken()`
- `MasterCrossLedgerVault.removeToken()`
- `MasterCrossLedgerVault.setNewMinimumOperationValue()`
- `MasterCrossLedgerVault.setNewMinSlippageProvider()`
- `CrossLedgerVault.addChain()`
- `CrossLedgerVault.removeChain()`
- `CrossLedgerVault.updateBridgeAdapter()`
- `SuperAdminControl.call()`

Events indicate major state changes. Hence, it might be useful for users to listen to certain events. Note that events do increase the gas costs slightly.

7.5 Redundant Operations

Informational **Version 1**

CS-APYF-007

There are multiple instances of redundant operations:

- `CrossLedgerVault.transferCompleted()` reads `bridgeAdapterToChainId`. However, this value is never used.
- `MasterCrossLedgerVault._startDeposit()` calculates the total score by reading the respective scores of all chains. Then, to calculate the respective deposits, the scores are read again.
- `BaseConcentratedLiquidityStrategy._redeem()` calls `_collect()` which is guaranteed to have been called before because of the harvesting mechanism.

7.6 Unreachable Operation

Informational Version 1

CS-APYF-008

`CrossLedgerVault._depositLocal()` handles the case where `assets == 0`. However, this case is unreachable. `_deposit()` returns if `amount == 0`. If the deposited amount to another chain is 0, then no deposit to that chain is actually executed.

7.7 Unused Variables and Function

Informational Version 1

CS-APYF-009

Some variables are unused and could be removed either from the parameter list of the respective functions or the function implementation:

- `sentTransfers` in `MasterCrossledgerVault._startDeposit()`
- `SlaveCrossLedgerVault._transferCompleted()`'s parameter `transferId`
- `CrossLedgerVault._blockingLzReceive()`'s parameter `srcLzChainId`

Note also that the function `WormholeBridgeAdapter.adjustAmount()` isn't used anywhere.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Asset Converters Can Always Be Frontrun

Note Version 1

Asset converters are used all across the protocol to swap between different assets.

It is important to note that at any point in time these swaps can be frontrun, and will especially be when the swapped amount is big enough.

Also note that the asset converter implements a slippage check to avoid big losses.

8.2 Asset Converters and Liquidity Provision

Note Version 1

Strategy vaults such as UniswapV3 can invest assets in LP positions of some pools. As the users deposit one specific asset, part of the deposited amount should be converted into another asset. For example, a user submits USDC only and a part of it is swapped to ETH for the two assets to be deposited together to an LP position in an ETH-USDC pool. For the required swap, there's no guarantee that the same ETH-USDC pool is not going to be used. This means that swaps needed to be performed can alter the price offered by the pool where the LP position is going to be opened. Furthermore, when the swapped amounts are big, for example during rebalances (calls to `readdLiquidity`), the deviation of the price of the pool can be significant. As these rebalances require two steps (`redeem` and `deposit`) which both check for a potential price deviation between pools and oracles, such deviation can block the deposit step and thus block the whole rebalance process.

8.3 Fees Accounting Is Based on Variable `pricePerToken`

Note Version 1

Fees accounting in `ApyFlow` is based on the variable `pricePerToken`. This variable can vary in both direction because of `ApyFlow`'s underlying strategies.

However, fees are only harvested if `pricePerToken` has increased since last harvesting. This mechanism makes it so that fees will only apply on the vault's total profit, but not on yields.

For example, let's say both `pricePerToken` and `lastPricePerToken` are equal to `1.1`. A strategy suffers a big loss, which decreases the price per token to `1`. Strategies still produce some yield, which make the `pricePerToken` grow back to `1.05` after a few days. No fees will be applied to this `0.05` growth in the price per token.

8.4 Fees Depend on the Converter

Note Version 1

In `ApyFlow.redeem()` fees are a portion of the `assets` sent to the user. However, `assets` are dependent on the `assetConverter`. Should the converter swap the assets with the maximum allowed slippage the fees earned by the system will be reduced.

8.5 Gas Limitations

Note **Version 1**

In the `ApyFlow` smart contract, each underlying vault in its `vaults` array is a `SingleAssetVault` which also contains an array of underlying vaults. In the case of a relatively high number of underlying vaults, gas costs can increase significantly even exceeding the gas limit. In that case, any deposit or redemption could block which could lead to blocking the entire system.

8.6 Liquidation Consequences

Note **Version 1**

In the `BaseHedgedConcentratedLiquidityStrategy` smart contract, a loan is taken on Aave to distribute the liquidity position risk evenly. During high price-volatility periods, a vault could therefore risk getting liquidated.

A liquidation would have multiple consequences:

- A sudden decrease in the price per token value.
- A potential risk of being unable to call `readdLiquidity()` due to the current price per token being smaller than `lastPricePerToken`, if the pool price returned out of the rebalance range after liquidation.

Note that yldr.com will configure hedged vaults so that the risk is very low and vaults should be monitored.

8.7 Slippage on Swaps Is Expected to Be Relatively Small

Note **Version 1**

With the current cross-ledger architecture, some token swaps will happen during operations through the asset converter smart contract. Note that this asset converter has a fixed slippage tolerance set and reverts if it is not respected.

Consequently, slippage tolerance should be properly set so that it isn't likely to block any cross-ledger action.

8.8 Supported Tokens

Note **Version 1**

The system only supports standard `ERC20` tokens without special behaviors, especially tokens with callbacks (`ERC777`) which would allow arbitrary code execution. More explicitly, tokens with two entry points should also be avoided.

Tokens with fees or any rebasing mechanism aren't supported.

8.9 Tokens Assumption

Note Version 1

Some parts of the code implicitly assume that some assets are the same, without ever checking if it is the case or handling the case where they are different.

The hedged concentrated liquidity strategy sometimes assumes that the `collateral` and `tokenToBorrow` tokens must be equal to the liquidity pool tokens. It also assumes that the `collateral` token is the same as the main asset of the vault.

These incomplete assumptions increase the code complexity and can lead to some complex errors.