Pods Finance Ethereum Volatility Vault Audit #1

OPENZEPPELIN SECURITY | DECEMBER 15, 2022

Security Audits

November 15th, 2022

This security assessment was prepared by **OpenZeppelin**.

Table of Contents

Summary

Type
: DeFi

Timeline
: From 2022-09-19
: To 2022-10-05

Languages
: Solidity

Total Issues
: 36 (24 resolved)

Critical Severity Issues
: 3 (3 resolved)

High Severity Issues
: 3 (3 resolved)

Medium Severity Issues
: 8 (4 resolved, 1 partially resolved)

Low Severity Issues
: 9 (5 resolved, 2 partially resolved)

Notes & Additional Information
: 13 (8 resolved)

Scope

We audited the pods-finance/yield-contracts repository at the 9389ab46e9ecdd1ea1fd7228c9d9c6821c00f057 commit.

In scope were the following contracts:

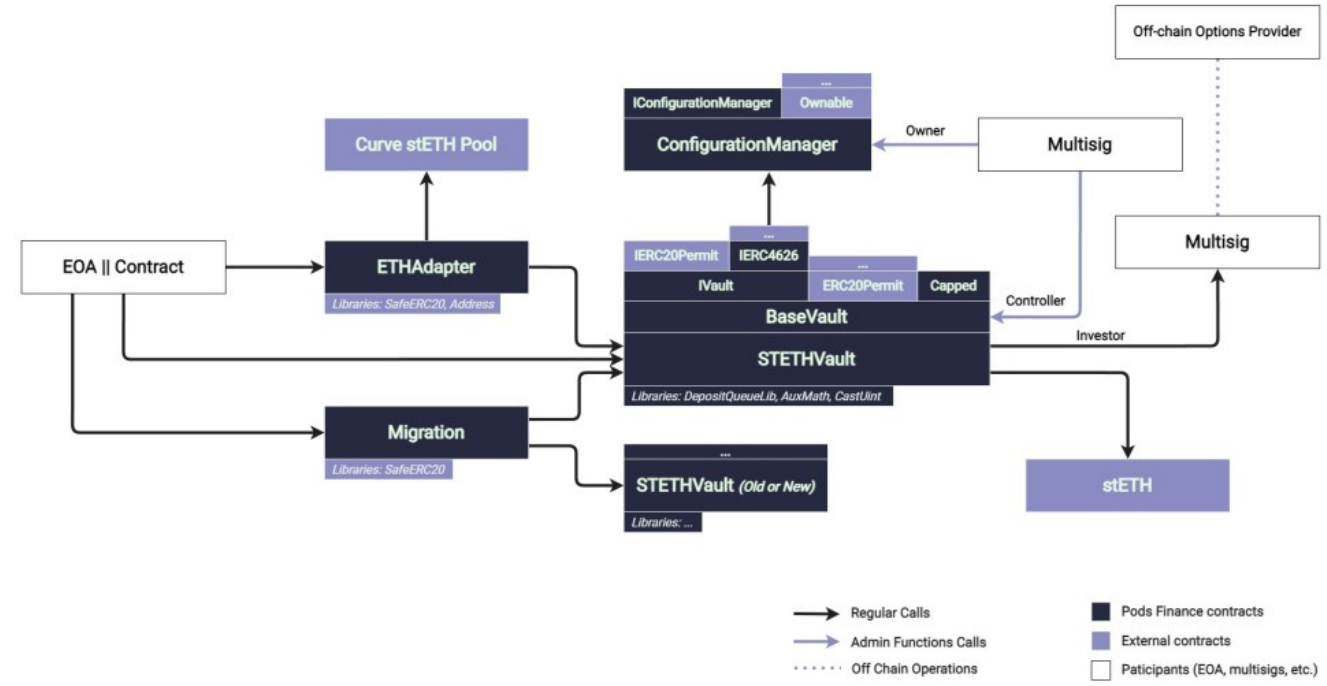
```
contracts/
├── configuration
│   └── ConfigurationManager.sol
├── interfaces
│   ├── IConfigurationManager.sol
│   ├── ICurvePool.sol
```

```
|   ├── AuxMath.sol
|   ├── CastUint.sol
|   └── DepositQueueLib.sol
├── mixins
|   └── Capped.sol
├── proxy
|   ├── ETHAdapter.sol
|   └── Migration.sol
└── vaults
    ├── BaseVault.sol
    └── STETHVault.sol
```

System Overview

The Pods Yield Vault is a one-click deposit investment product that features an options strategy to generate principal-protected returns when the ETH market is volatile. The underlying technique uses a portion of the yield from stETH assets to invest in the same number of call and put ETH options in an off-chain market. Returns from these investments will be given to the participants who provided their yield to the investor.

The following diagram gives an overview of the system:



stETH Volatility Vault

The Ethereum Volatility Vault, `stETHvv` for short, is a tokenized vault on the Ethereum mainnet, conforming to the EIP-4626 standard. It integrates with the Lido `stETH` token as the single underlying asset. Users deposit the underlying asset into the vault in return for the `stETHvv` vault shares to participate in the investment strategy.

The investment strategy is executed in rounds. After a new round starts, users can join the deposit queue, where the users' underlying assets will be transferred to the vault, thus starting to generate yield to the vault. However, `stETHvv` vault shares will not be minted to the user until the queued deposit is processed after the round ends.

The user can withdraw their queued deposit before it is processed, but any `stETH` yield generated by the queued deposit will not be accessible. Before the round ends, users with `stETHvv` vault shares can withdraw their share of assets for a fee.

When the round ends, only a portion of the `stETH` yield, not the principal, will be transferred to a centralized `investor` account, where the yield is manually used to participate in the off-chain

After the round ends, deposits to the queue and withdrawals from the vault are temporarily paused while processing of queued deposits is enabled. Anyone can process any queued deposit by specifying a valid start and end index of the queue. Each processed deposit will mint new `stETHvv` vault shares, as computed based on the total assets in the vault. Any unprocessed deposit will remain in the queue, which can be either processed or withdrawn in the next round.

When the next round starts, new deposits and withdrawals to the queue may resume, while queued deposits can no longer be processed.

ConfigurationManager

Parameters such as the vault controller, the percentage of commission for withdrawals, and the cap of shares that can be issued are established in this contract for the different vaults of the protocol. One particularity of this contract is that these values are set through a function that allows storing any name-value pair for a specific target on a nested mapping. This allows more values to be added in the future without the need to replace the contract but adds more complexity to managing contract storage.

It is worth mentioning that there is an exception where an exclusive function was defined to establish if a vault can be used as a migration destination. It has its own setter and getter.

ETH Adapter

In order to allow vault deposits to be made directly with ETH, this contract acts as an intermediary so that users can participate in the protocol without needing to perform the swap themselves. This contract swaps ETH for stETH through a Curve Finance pool. After the exchange, the ETH adapter deposits the resulting tokens to the vault on behalf of the user within the same transaction.

Future Migration

In the case of an upgrade, a `stETHvv` vault shareholders can call `migrate()` on the current vault contract to redeem their shares from this vault and join the deposit queue in the new vault. Also, users can leverage the migration contract, an implementation that has already set the origin and destination vaults, to join the deposit queue in a new vault. In addition to

Permit

In order to fulfill its value proposition of being a one-click deposit investment product, Pods Finance team implements permit functionalities on deposits and other key vault functions. With this, users can sign a permit off-chain and then use the functions with permit integrations to perform the approval in the same transaction.

Privileged Roles

Some privileged roles exercise powers over the vault and periphery contracts:

- `Owner` of the `ConfigurationManager`
  - Can add, remove and modify system parameters such as the cap, the controller, and others.
  - Can allow a vault to be a destination in case of a migration.
- `Controller` of the `vault`
  - After a previous round ends, the `controller` can start a new round anytime.
  - Anyone can start a new round if a week has passed since the last round ends. After a round starts, only the `controller` can end a round.
  - It is the recipient of the fees collected.
- `Investor` of the `vault`
  - When a round ends, the investor account collects a portion of the yield from the deposited assets and transfers any profit obtained from the previous round of investment to the vault. Any assets in the investor account are visible to all. The investor account holds all the assets used for the off-chain option investment strategy.

Findings

Here we present our findings.

Critical Severity

DepositQueue can become permanently locked

underlying remove function in order to restructure the `DepositQueue`.

The issue is that the `remove` function performs gas-intensive operations that drastically limit the upper bound on the queue size, which we conservatively estimate to be in the magnitude of a few thousands. With a large enough queue, any transaction attempting to refund or process even a single deposit will exceed the block gas limit of 30,000,000 gas. This could occur due to organic user growth, or a malicious actor may choose to send dust amounts of stETH to the vault from different addresses to take up positions in the queue. Without the ability to process or refund users, all current and future deposits made to the vault would be locked in the `DepositQueue`.

Consider accounting for the gas usage of all queue operations and ensuring they cannot reasonably exceed the block gas limit. Some recommendations to consider:

- Refactor the `DepositQueue` so that operations such as `remove` can be performed in significantly less (and preferably constant) gas
- Introduce a maximum capacity to the `DepositQueue` size to guarantee it can be processed each round
- Apply a minimum investment requirement to deter attackers from spamming the queue with dust amounts of stETH
- Use a different data structure for the `DepositQueue`, such as an EnumerableMap, to reduce code maintenance overhead and ensure necessary operations can be performed in constant gas

Update: *Fixed in commit* `e05f165`.

Miscounted assets breaking system invariants

When processing queued deposits, the amount of assets that have been processed are accounted for in the `processedDeposits` variable and used to calculate the amount of shares to be minted. This variable is reset to 0 at the beginning of each round.

However, each time the `processQueuedDeposits` function concludes, the processed amount will be removed from `totalIdleAssets`, hence increasing the `totalAssets`. When the queue is processed the next time, the `totalAssets` value already contains the

This penalizes depositors by minting fewer shares if they are not included in the first call of the `processQueuedDeposits` function. This issue is particularly severe if each deposit is processed individually. Hence it breaks the system invariant that the number of splits should not modify the result when processing the deposits.

Additionally, this means users not processed in the first round may receive fewer assets than they deposited when they redeem their shares. This breaks another important system invariant that 100% withdrawal in stETH should always be equal to or higher than the initial deposit amount.

Consider correcting the total asset calculation to ensure the processed amount is accounted for accurately.

Update: *Fixed in PR#54, with commit* `3ad9b76` *being the last one added.*

The vault can be drained one share at a time

During the withdrawal process, users can specify the amount of assets to withdraw, which is then rounded down to shares.

When the asset amount specified by the user is less than the minimum amount that can be converted to a unit share, the `shares` argument is zero in the internal withdraw function but the `assets` argument is not. Hence, with zero shares, all internal calls can succeed and a non-zero amount of asset token will be transferred out to the receiver without burning any shares. This process can be repeated many times to drain the entire vault. The attack can also be executed with any asset amount by burning a rounded-down amount of shares and extracting the excess assets.

Since the vault is expected to become more valuable over time due to its yield strategy, this could lead to a profitable attack when one share is worth more than the cost.

Consider rounding up the shares for a given amount of assets during withdrawal.

Update: *Fixed in PR#46, with commit* `5ac5e3c` *being the last one added.*

The controller should be able to call the `endRound()` function any time after a round starts to remove a portion of the accrued interest from the vault to the `investor` account for the execution of investment strategies. In the `_afterRoundEnd` hook, the `totalAssets()` is compared to the `lastRoundAssets` to compute the accrued interest. When a user withdraws assets before the `endRound`, the `lastRoundAssets` is updated in the `_beforeWithdraw()` hook, where the withdrawn shares are rounded down to the corresponding asset value, which is subsequently subtracted from the `lastRoundAssets`.

This may cause the `lastRoundAssets` to be more than the `totalAssets()` when a user calls the `withdraw` function. This can result in a revert due to underflow. In such a case, the controller cannot end a round and must wait for the accrued interest to accumulate enough value.

Consider keeping track of the withdrawn asset directly. Also, consider making sure the key system functionality is not impeded by users' actions.

Update: *Fixed in PR#73, with commit `b258ca9` being the last one added.*

Reentrancy risk in depositing to the queue

The internal `_deposit` function handles user deposits, transferring a specified amount of `stETH` from `msg.sender` to the vault. Before moving the funds, it adds the deposit to the queue, which is processed later by the `processQueuedDeposits` function.

As the underlying token could have hooks that allow the token sender to execute code before the transfer (e.g., ERC777 standard), a malicious user could use those hooks to re-enter the `deposit` function multiple times.

This re-entrancy will result in an increment in the receiver balance on the queue, even though this balance will not correspond to the actual amount deposited into the vault.

In the current implementation, the `_deposit` function in the `BaseVault` contract is overridden by the implementation in the `STETHVault`, which has the correct order of operation. However, the `BaseVault` is likely to be inherited by future vaults, so it is crucial to have the correct `_deposit` implementation in this contract in case it is not overridden.

which already has this solution implemented.

Update: *Fixed in PR#41, with commit* `2ffcb1e` *being the last one added.*

Rounding up in minting shares

When processing the queued deposits, shares are minted to the receiver in the queue according to the amount of assets deposited. However, the amount of shares minted is always rounded up. This means that one can always receive 1 vault share with a 1-wei deposit.

As the vault is expected to be increasing in value from yield rewards, 1 vault share will be worth more than 1 wei asset eventually. A malicious user can spam the deposit queue with 1-wei deposit from many accounts to get 1 share each and then redeem them for more assets when each share is worth more.

Consider rounding down when minting vault shares.

Update: *Fixed in PR#46, with commit* `5ac5e3c` *being the last one added.*

Medium Severity

Non-existent permit function

The vault implements a `mintWithPermit` and `depositWithPermit` function intended to allow users to transfer assets to the vault in a single transaction. However, the vault's underlying asset is intended to be stETH which does not have a `permit` function. Currently, any user who tries to perform a `mintWithPermit` or `depositWithPermit` will have their transaction reverted due to the stETH contract's fallback function.

Consider removing the `mintWithPermit` and `depositWithPermit` functions. We note that wstETH does have a permit function for future considerations.

Update: *Acknowledged, and will not fix. Pods Finance team's statement for this issue:*

> *We decided not to fix this issue because we may implement assets similar to LIDO as the yield source where they may have permit functionality (aTokens, for instance).*

sending and receiving ETH instead of stETH. The adapter achieves this by converting ETH and stETH through a curve pool and then forwarding interactions to and from the vault. In the course of a normal withdrawal or redemption transaction, the `ETHAdapter` will pull the funds out of the vault before passing them on to the designated receiver. During the moment the `ETHAdapter` is holding the funds, it first converts all of its stETH to ETH, and then sends its entire ETH balance to the receiving address.

The issue is that the `ETHAdapter` sends its full balance to the receiver each time, meaning any ETH or stETH that is mistakenly sent to it can be drained by any user who performs a withdrawal or redemption on the `ETHAdapter`. This is exacerbated by the fact that the vault is passed in as a parameter, potentially allowing a user to perform withdrawals and redemptions without interacting with the actual stETH vault.

Consider transferring out the exchanged balance from the curve pool to the receiver instead of the entire balance of the `ETHAdapter`. Also consider implementing a rescue or sweep function to allow the recovery of funds that are accidentally sent to the `ETHAdapter`.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *For now, we do not want to take action in case of funds sends by mistake to our contract. We see this as a low-priority issue.*

Incorrect calculations

We found the following instances of incorrect calculations in `view` functions that are not currently called internally:

- In the `previewWithdraw` function, the `DENOMINATOR` over `invertedFee` is always bigger than 1 when the fee is non-zero. Hence, the final returned shares are always an overestimate. Further, the `withdrawFeeRatio` is multiplied to `shares` instead of `assets` as in other instances such as `_getFee`. Consider correcting the withdrawal fee calculation.
- The calculation in the `assetsOf` function over-estimates the actual commitment by an additional `committedAssets`. Consider removing the extra component.

Math library is vulnerable to shadow overflow

The `AuxMath` library is a custom auxiliary math library that performs multiplication and division with rounding specifications.

The implementation of `mulDivUp` and `mulDivDown` first compute the multiplication of a `uint256` with a `uint256` without taking into account the possibility of an overflow in the product. The product could overflow into a `uint512` in the multiplication step even if the result were to fit into a `uint256` after the subsequent division. Hence the current implementation will not give the right result in such a case.

Consider using OpenZeppelin's Math library which implements the `mulDiv` function that is developed especially considering these scenarios and is widely accepted in the ecosystem.

Update: *Fixed in commit 2a8c58a.*

Maximum mintable and depositable amounts returned are incorrect

The amount of vault shares available for minting is limited by the `availableCap`. Similarly, the `availableCap` also limits the amount of assets that may be deposited and then later converted into vault shares for the depositor. The `maxMint` and `maxDeposit` functions do not account for the `availableCap` and instead return the max `uint256` value when called. In order to conform to the `EIP4626` standard, these functions must return the real amount that can be minted or deposited.

Consider changing the `maxMint` and `maxDeposit` functions to return a value that accounts for the `availableCap`.

Update: *Fixed in PR#42, with commit `3159de5` being the last one added.*

Refund does not restore the cap

When a user deposits into the vault by joining the deposit queue, the corresponding shares, yet to be minted, are deducted from the spending cap immediately. The spending cap is restored only at withdrawal when the shares are burned.

large amount of assets could repeatedly deposit and [illegible] to reach the cap limit and stop other eligible users from joining the queue.

Note that the `spendCap` variable cannot be manually restored, but the owner can reset the cap to a higher value to unlock the deposit.

Consider accounting for the available cap during refunds from the queue.

Update: *Fixed in commit* `4a2e475`.

Refund can be over-credited in a negative yield event

Deposits added to the queue are point in time stETH balance amounts. The stETH token balance rebases regularly to account for yield, and in the event of slashing, may be subject to a negative yield. In the event that a stETH token rebase is negative between the time a user deposits and calls for a refund, the vault will over credit the user by the rebase difference.

Consider handling the deposits in the queue in shares instead of balances to account for rebase changes on refunds.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *Although we agree with the issue, we won't prioritize it right now because of timing. In case of a slashing event, we can refund the vault, transferring funds directly to the contract.*

Parallel share cap setting

The configuration manager determines the maximum vault shares available for minting. When setting a new cap through the `setCap` function, the target contract address is checked for the zero address, and a `SetCap` event is emitted.

However, the underlying `setParameter` function can be called directly to change the maximum vault shares available. This will bypass the zero address check and emit a `ParameterSet` event instead of the expected `SetCap` event.

Consider ensuring consistency between the two mechanisms in setting the share cap. Depending on the desired outcome, this could involve removing the `setParameter` function and adding

Low Severity

Gas inefficiencies

There are many instances throughout the codebase where changes can be made to improve gas consumption. For example:

- `public` functions might consume more gas than external functions. This is because with public functions, the EVM copies inputs (especially dynamic-sized arrays) into `memory`, while it reads from `calldata` if the function is `external`, which is cheaper. Throughout the codebase there are `public` functions like `mintWithPermit`, `convertToSTETH`, `convertToETH`, `decimals`, `symbol`, name and `assetsOf` that might be declared as `external`. Consider reviewing the entire codebase for more instances.
- The function `toAddress` has a conditional statement that returns the zero address if the value passed is equal to zero. This additional check increases the gas consumption and does not add any benefit even if the provided value is equal to zero.
- Initializing a variable to its default value causes unnecessary gas expense. Consider correcting such instances in `processedDeposits`, `isProcessingDeposits`, `roundAccruedInterest`, `endSharePrice` and the while loop counter inside the `remove` function.
- The `for` loop inside the `refund` function does not cache the result of `depositQueue.size()` and perform unnecessary operations on each iteration. The same issue occurs in the last while loop of the `remove` function.
- Performing calculations for values that will not change is suboptimal. Some examples are: `name`, `symbol` and the operation `10**sharePriceDecimals`.
- Use shorter string messages for required statements or switch entirely to a custom errors implementation to save on deployment cost and failed execution gas cost.

When performing these changes, aim to reach an optimal tradeoff between gas optimization and readability. Having a codebase that is easy to understand reduces the chance of future errors and improves community transparency.

The name attribute in the constructor of the `BaseVault`, passed to the `draft-ERC20Permit` contract does not match the name returned from the `STETHVault` implementation. All vaults that use the same underlying asset and inherit from `BaseVault` will have the same `hashedName` when calculating the domain separator by the EIP712 standard. Although this does not represent a risk, it is recommended that the name sent to the `ERC20Permit` match the ERC20 token name.

Also, the `name` and `symbol` defined in the `BaseVault` constructor are inconsistently overridden in the `STETHVault` implementation.

To avoid complications and improve code clarity, consider passing both the name and symbol from the `STETHVault` contract as parameters to maintain consistency. This also saves calculating the name and symbol at every call, thus saving more gas.

Update: *Fixed in commit* `61a9996`.

Misleading comments

Comments are meant to be helpful and describe the intent of the code block. However, in multiple parts of the codebase, comments are not consistent with the code that is written or do not add any value to the reader. These are some examples:

- The function `_deposit` in both the `STETHVault` and `BaseVault` contracts does not create any shares even though the comments say so.
- The description of each parameter on `setParameter`, `getParameter` and `getGlobalParameter` does not add any value to the reader.
- The `target` parameter on `setParameter` has a wrong description.
- The function `_spendCap` has the same description as `availableCap`.

Consider correcting the comments to not mislead users or developers and write comments that add value to the reader. This will improve code clarity and consistency between docstrings and contract implementations.

Throughout the codebase, there are several files that do not have docstrings. For instance:

- The `ConfigurationManager` contract has no docstrings for any of its state variables.
- The `IConfigurationManager` interface does not have any docstrings.
- Although the `ICurvePool` interface is for an external contract, docstrings need to be added to explain the interface purpose and why each function is used within the codebase.
- The `IERC4626` interface has no docstrings for the interface description and also for any of its events.
- The `IVault` interface has no docstrings for the interface description and also for any of its events and errors.
- The `AuxMath` library does not have any docstrings.
- The `CastUint` library has no docstrings for the library description.
- The `DepositQueueLib` library just has some inline comments inside the `remove` function.
- The `Capped` contract has no docstrings for the contract description, state variables, constructor, and its custom error.
- The `ETHAdapter` contract just has some inline comments for the receive function.
- The `Migration` contract does not have any docstrings.
- The `BaseVault` contract has no docstrings for some of its state variables (`configuration`, `_asset`, `currentRoundId`, `isProcessingDeposits`, `EMERGENCY_INTERVAL`, `processedDeposits`, `_lastEndRound` and `depositQueue`), the constructor, and the functions `depositWithPermit` and `mintWithPermit`.
- The `STETHVault` contract has no docstrings for some of its state variables (`sharePriceDecimals`, `lastRoundAssets`, `lastSharePrice`, and `investor`) and the constructor. Adding comments in the internal function `_afterRoundEnd` can be beneficial for easy understanding of its purpose.

This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security but also correctness. Additionally, docstrings improve readability and easy maintenance.

public, should be clearly documented as well. When writing documentation, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *We didn't prioritize this because of time constraints.*

Processing order alters queue indexes

Any user may process deposits in the queue after a round ends. After deposits are processed, the section of the queue they were held in is removed. This means that any user who processes deposits will be changing the size and indexes of the queue for all others. This may lead to reverted transactions if an index that was previously thought to be in range no longer exists in the queue. It may also lead to some processing of unintended indexes and others missing their targeted indexes in the queue.

Consider having additional validations on the queue size and indexes to ensure the correct transactions are executed without going out of bounds.

Update: *Fixed in commit* `e05f165`.

Refunds are subject to high gas costs and a potential DoS

Refunds are processed by searching the entire queue front to back for a user's position. Currently, the gas price to refund increases linearly based on the user's position in the queue. Users that have joined the queue later in the round will incur a more significant gas penalty and, in an extreme case, could be prevented from refunding.

Consider keeping track of a user's queue position alongside other attributes for easier retrieval and update.

Update: *Fixed in commit* `e05f165`.

Remove function does not revert if `startIndex` is greater than or equal to `endIndex`

continues. This could cause `processQueuedDeposits` to fail silently when the indexes are not valid.

Although it does not currently cause any errors other than spending gas unnecessarily, consider rolling back the transaction if the condition is met to avoid unwanted behaviors.

Update: *Fixed in commit* `e05f165`.

Migration risks

The system allows users to migrate their shares to a new vault with the same underlying asset. There are a few risks regarding the current migration mechanism design.

- **Unclear migration path**: Currently, two migration mechanisms exist. A user can either call the `migrate` function in the old vault or through the exclusive `Migration` contract. There is no clarity on which route is preferred as both essentially perform the same functionality. Consider removing duplicate actions that can lead to user confusion.
- **Incomplete migration**: In both migration routes, users first redeem their shares from the old vault and then re-deposit the assets into the new vault. If the queueing system works the same way in the new vault, this only lets the user join the deposit queue. Hence the migration is not complete until the queued deposits are processed with the shares minted. Consider documenting this to make it clear to users who migrate.
- **Migration re-usability:** The `Migration` contract sets both the old vault and the new vault addresses as immutable variables in the constructor. This limits its use case to one single migration. It would not be reusable across multiple vaults with the same underlying assets and different investment strategies.
- **Not robust against trapped funds**: When assets are present in the `Migration` contract due to direct transfers or mistaken withdrawals, the entire balance of the contract will be deposited to the new vault by the next user. This design is not resistant to genuine user mistakes and prohibits recovering user funds. Consider using the returned value from the `redeem` function to migrate the correct amount of assets and establishing a sweep or rescue function for funds locked in the contract.

Update: *Partially fixed in commit* `d97672d`.

the vault to the same `investor` account. Instead of making two separate transactions, one can compare the two values and reach the final balance in one transaction.

Consider simplifying the code for clarity and readability.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *We actively preferred to leave the code in the back-and-forth way, so it's easier to understand the flow of funds of the operations.*

Notes & Additional Information

Magic numbers are used

Although constants are generally used correctly throughout the codebase, there are a few occurrences of literal values being used with unexplained meaning. For example, the following blocks use hardcoded values:

- In the constructor of the `ETHAdapter` contract, the pool's `coins` method is called with arguments `0` and `1`, with no explanation.
- In the functions `convertToETH` and `convertToSTETH`, the pool's `get_dy` method is called with arguments `0` and `1` with no explanation.
- In the `deposit` function, the pool's `exchange` method is called with arguments `0` and `1`, with no explanation.

To improve the code's readability and facilitate refactoring, consider defining a constant for every magic number, giving it a clear and self-explanatory name. Consider adding an inline comment explaining how they are calculated or why they are chosen for complex values.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *We didn't prioritize this because of time constraints.*

Multiple conditions for error handling statements

- In the constructor of the `ETHAdapter` contract there is a `require` statement that checks if coin 0 from the pool is equal to the `ETH` address and if coin 1 is equal to `stETH` address. If one of the conditions is not met, there is no error message to indicate why the transaction was reverted.
- The `migrate` function in the `BaseVault` contract has a conditional block that checks if the vault is not allowed and if the destination vault does not have the same underlying asset as the current vault. Although the custom error that is thrown is not wrong, it might be better if these two validations are separated to return more specific error messages.
- The `onlyRoundStarter` modifier has a custom error block that checks if less than a week has passed since the last end round and if the caller is different from the controller. If the conditions are met, the error returned is that the caller is not the controller. Consider separating both conditions in order to throw a more specific error message.

To simplify the codebase and to raise the most helpful error messages for failing blocks, consider having a single `require` statement with an appropriate error message or a single custom error block per condition.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *We didn't prioritize this because of time constraints.*

Naming issues hinder code understanding and readability

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:

- Function `withdrawFeeRatio` can be changed to `getWithdrawFeeRatio`. The current name suggests that the action is the withdrawal of fee ratio but what it actually does is to query the `WITHDRAW_FEE_RATIO` value.
- Error `ConfigurationManager__InvalidCapTarget` can be changed to `ConfigurationManager__TargetCannotBeTheZeroAddress` or something similar, since the only invalid target is the zero address.

Update: *Fixed in commit* `88ce562`.

lead to conflicts between names defined locally and the ones imported. This is especially important if many contracts are defined within the same Solidity files or the inheritance chains are long. For instance, but not limited to:

- Lines 5-14 of `BaseVault.sol`
- Lines 5-9 of `EthAdapter.sol`

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *We didn't prioritize this because of time constraints.*

Outdated Solidity version

All contracts within the codebase use compiler version `0.8.9`, an old version with various bugs that have been fixed in later releases.

Consider taking advantage of the latest Solidity version to improve the overall quality and security of the codebase.

Update: *Fixed in commit* `93956a9` *in PR#66.*

State variable visibility not explicitly declared

Throughout the codebase, there are state variables that lack explicitly declared visibility. For instance:

- `CAP` in `ConfigurationManager.sol`
- `ETH_ADDRESS` in `ETHAdapter.sol`
- `STETH_ADDRESS` in `ETHAdapter.sol`
- `from` in `Migration.sol`
- `to` in `Migration.sol`

Update: *Fixed in PR#64, with commit* `a3eb939` *being the last one added.*

Style inconsistencies

Style inconsistencies can be seen throughout the codebase. For example:

- Some functions use `require` statements for error handling and others use custom errors. In general custom errors are preferred over `require` statements.
- The naming convention for events is not clear. Some use, like `StartRound`, the function's name with the first letter capitalized, and others, like `FeeCollected`, describe the action that was executed. Consider following only one convention on the entire codebase.
- The constant `investorRatio` does not use all capital letters with underscores separating words but all other constants in the codebase do. Consider changing it to `INVESTOR_RATIO`.

Update: *Fixed in commit* `cf6ae4d`.

Typographical errors

Across the codebase, there are some typographical errors in the comments. Some examples are:

- `ConfiguratorManager` should be `ConfigurationManager`.
- `MAX_WITDRAW_FEE` should be `MAX_WITHDRAW_FEE`.
- splitted should be `split`.

Consider correcting the above and any further typos in favor of correctness and readability.

Update: *Fixed in* `PR#41` *commit* `a515964` *and* `PR#69` *commit* `4ec70cb`.

Unnecessary arguments

We found some internal functions with unused arguments in the implementation, hence are unnecessary to be defined in the base contract. For example:

- The `_afterRoundStart(uint256)` function does not require an argument in the implementation but is defined with an argument in the base contract.

Consider removing unnecessary arguments for clarity and readability.

Update: *Fixed in commit* `f100925`.

Unnecessary implementation

The following instances of internal virtual functions are implemented in `BaseVault.sol`, which are later overridden by a different implementation in `STETHVault.sol`.

- The `_withdraw` function is overridden by the same implementation.
- The `_deposit` function is overridden by a different implementation.

Consider removing unnecessary implementations for readability and maintenance.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *We didn't prioritize this because of time constraints.*

Unused Code

The global address and the `getGlobalParameter()` function of the `ConfigurationManager` contract are supposed to manage global configurations that do not depend on a specific vault. However, the purpose of those entities is unclear, as they are not used anywhere in the codebase.

To avoid confusion and favor explicitness, consider documenting the purpose of these functions and variables. If they are not expected to be used, consider removing them from the codebase to reduce the code's size and attack surface.

Update: *Acknowledged, will not fix. Pods Finance team's statement:*

> *We didn't prioritize this because of time constraints.*

Unused imports

Throughout the codebase imports on the following lines are unused and could be removed:

Consider removing unused imports to avoid confusion and improve the overall readability of the codebase.

Update: *Fixed in commits* `b702189` *and* `89869ad`.

Unused named return variable

Throughout the codebase, there are several occasions where named return variables are declared but not used. For instance:

- `newShares` on `migrate` and `migrateWithPermit` functions
- `stETHAmount` on `convertToSTETH`
- `ethAmount` on `convertToETH`
- `depositedAssets` on `_deposit` on both `STETHVault` and `BaseVault`
- `roundId` on `startRound`
- `shares` on `maxMint`, `maxRedeem`, `deposit`, `convertToShares`, and `previewDeposit`
- `assets` on `maxWithdraw`, `maxDeposit`, `convertToAssets`, and `previewMint`
- `effectiveAmount` on `_stETHTransferFrom`

Consider adding those variables to function implementations or remove them.

Update: *Fixed in commit* `1228184`.

Conclusions

Three critical and three high-severity issues are found among various lower-severity issues. The system documentation and diagrams provided by the Pods Finance team made it easier for the auditors to assess and comprehend the code. Throughout the audit, the team has been responsive and active in responding to questions and suggestions from the auditors.

Given the number of important issues raised, we believe there could be more undiscovered issues, and we anticipate a significant refactor to the codebase. If this turns out to be the case, we highly recommend another round of audits on the new code.

initially reported; however, another critical issue was discovered during the fix-review process. OpenZeppelin worked with the Pods Finance team to address this issue, though we noted that the fix-review process was limited in assessing security issues within the significant changes made to the codebase. OpenZeppelin strongly recommended the Pods Finance team obtain a full code re-audit and the Pods Finance team is proceeding with that.

Appendix I: Monitoring Recommendation

With the goal of providing a complete security solution, we have identified several actions that might benefit from on-chain monitoring to provide useful alerts and insights about on-chain protocol activities. Here are some example scenarios that would benefit from monitoring:

- `Deposits` to the `vault` with dust or large amounts. This could represent a potential threat to manipulate vault share calculations.
- Privileged role activities such as those performed by the `Owner` of the `ConfigurationManager`, the `Controller` of the `vault`, and the `Investor` of the `vault`. Consider monitoring events related, but not limited to, ownership changes, vault configuration modifications, and transfers initiated by the `Investor` to somewhere other than the `vault` as these may signal key compromise of a privileged role.

Appendix II: Issue noted during fix-review process

Critical Issue: Malicious user can manipulate shares calculation and steal deposits

As part of the fixes developed and provided during the fix-review process, it was noted that the current queue setup could potentially be exploited by Maximum Extractable Value (MEV) searchers if the `endRound` transaction is sent through the public mempool.

In the scenario when the MEV searcher has sufficient liquidity, the following strategy can be viable.

- Deposit one wei to the queue before `endRound`.
- Back-run the `endRound` transaction with a transaction that
    - direct transfers an amount equal to the max of the queued amounts plus one wei.

In this way, the MEV searcher gets one share while everyone else gets zero shares due to the rounding in the division. Since no deposit or withdrawal is possible during the processing period, the MEV searcher waits until either the next `startRound` or one week has passed. Then the searcher can redeem its shares and remove the entire liquidity from the vault. We note that the searcher may need to back-run the `startRound` transaction to ensure that its one share is not diluted by further large sum deposits. Now, the vault is back to zero shares, zero liquidity and the same attack can be repeated.

In case the MEV searcher does not have enough liquidity, a milder version of the same strategy can be carried out by processing only part of the queue with the correct ordering.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits