PUBLIC

Code Assessment

of the Pendle V2 Core Smart Contracts

August 15, 2024

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Pendle Finance with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Pendle V2 Core according to Scope to support you in forming an opinion on their security risks.

Pendle Finance implements a yield tokenization platform along with an interest rate market. The products are built on top of each other and use the implementations of the standardized yield standard as a base layer.

The most critical subjects covered in our audit are functional correctness, asset solvency, arithmetic operations and oracle safety.

Generally, functional correctness is good. However, note that there are some low-severity issues regarding functional correctness. Security regarding the remaining subjects is high.

The general subjects covered are gas efficiency, trustworthiness, error handling and specification. Security regarding all the aforementioned subjects is good. However, specifications could be improved, see Initial Liquidity Mismatches Whitepaper and Mismatches With EIP-5115.

In summary, we find that the codebase provides a good level of security. Also, note that the security of SYs is highly dependent on the more derived implementation which was out of scope. Further, note that the scope only includes the base SY implementation, PY V1 and markets V1. Please see Assessment Overview, Trust Model and Roles, and Notes.

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	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	9
• Code Corrected	2
• Code Partially Corrected	1
• Risk Accepted	2
• Acknowledged	4

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 Pendle V2 Core repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Public Repository

V	Date	Commit Hash	Note
1	05 September 2023	979d55419b65b2565964021f50da25b36acd6d32	Initial Version

Private Repository

V	Date	Commit Hash	Note
1	05 November 2023	e01d169859c92b17a70490adefa0b75c754bf9d0	Second Version
2	25 November 2023	48a7abc1a04c5cea927a3c446b7be0f8cc16839f	Third Version

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

The files in scope are:

```
./core/YieldContracts/PendlePrincipalToken.sol
./core/YieldContracts/InterestManagerYT.sol
./core/YieldContracts/PendleYieldToken.sol
./core/YieldContracts/PendleYieldContractFactory.sol
./core/erc20/PendleERC20Permit.sol
./core/erc20/PendleERC20.sol
./core/libraries/MiniHelpers.sol
./core/libraries/TokenHelper.sol
./core/libraries/Errors.sol
./core/libraries/math/LogExpMath.sol
./core/libraries/math/Math.sol
./core/libraries/BoringOwnableUpgradeable.sol
./core/libraries/ArrayLib.sol
./core/Market/OracleLib.sol
./core/Market/PendleMarketFactory.sol
./core/Market/PendleGauge.sol
./core/Market/PendleMarket.sol
./core/Market/MarketMathCore.sol
./core/StandardizedYield/SYBase.sol
./core/StandardizedYield/SYBaseWithRewards.sol
./core/StandardizedYield/PYIndex.sol
./core/StandardizedYield/SYUtils.sol
./core/StandardizedYield/implementations/PendleWstEthSY.sol
./core/RewardManager/RewardManager.sol
./core/RewardManager/RewardManagerAbstract.sol
./oracles/PendleLpOracleLib.sol
```

```
./oracles/PendlePtOracle.sol
./oracles/PendlePtOracleLib.sol
```

2.1.1 Excluded from scope

All files not mentioned above are not in scope. Further, we exclude all the external systems, assume standard ERC-20 tokens, expect honest contract owners, and assume the more derived contracts to be fully correct. Please see Trust Model and Roles and Notes for more details. Note that only the SY base, PY V1, and the market in consideration with these contracts is in scope.

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.

Pendle Finance offers wrappers around yield-bearing tokens that adhere to EIP-5115 (called SY). A yield tokenization system is built on top of the SY so that the yield and principal components (PT and YT - called PY short) are separated into two tokens. Additionally, a market is implemented that allows redeeming PT before expiry for SY. Ultimately, a fixed yield mechanism is implemented while it can also be used to speculate on yield.

2.2.1 Reward Manager

The reward manager consists of two abstract contracts that are inherited by several contracts in the system. This section outlines the reward mechanics.

The `RewardManagerAbstract` implements the internal functions `_updateAndDistributeRewards()` and `_updateAndDistributeRewardsForTwo()` that first updates and retrieves reward indices through `_updateRewardIndex()` (up to the contract inheriting from this). Given that index, it checkpoints the provided user(s) with the current index so that the formula below describes the amount of reward tokens owed to the user:

$$accrued_{user,\,token} = rewardShares_{user}(rewardIndex_{current} - rewardIndex_{user})$$

Note that the formula depends on the shares with which the user is claiming rewards (`_rewardSharesUser` which is not implemented). As a consequence, the indices should depend on the number of total shares so that the invariants that the sum of $accrued$ should not exceed the total rewards held. Additionally, this library assumes that the index is monotonically increasing.

Note that the contract leaves out how (abstract `_redeemExternalReward()`) and when rewards are redeemed, and how (abstract `_doTransferOutRewards()`) and when rewards are sent out.

The `RewardManager` contract further defines the reward mechanism defined above. Namely, it defines the indexes more concretely as

$$index_{t_i} = \sum_{i \in [0 \ldots t_i]} \frac{rewards_i}{totalRewardShares_i}$$

Hence, the global index is defined as the sum of all rewards per reward share and thus the `_updateRewardIndex()` has been implemented accordingly. A user's distributed rewards will hence be

$$\sum_{i \in [t_i + 1 \ldots t_j]} \frac{rewards_i}{totalRewardShares_i}$$

where t_i is the user's checkpointed index and t_j is the global index.

The assumption of the accrued computation in the abstract manager is respected as long as the following holds. The sum of reward shares must not exceed the total reward shares (`_rewardSharesTotal()` abstract method) and before every reward shares balance change occurs, the rewards must be updated for the users.

2.2.2 Standardized Yield (base)

The Standardized Yield (SY) contracts integration adapters are ERC-20 tokens wrapping a yield-bearing token (`yieldToken`) and have a standardized interface implementing EIP-5115. More specifically, users can `deposit()` valid input tokens according to `getTokensIn()` and `isValidTokenIn()`, to generate the `yieldToken` so that an equal amount of SY tokens can be minted. Similarly, users can `redeem()` their SY token for valid output tokens according to `getTokensOut()` and `isValidTokenOut()`, to convert the `yieldToken` to the desired output token. Note, that the input and output token could be the `yieldToken` itself (e.g. both USDC and cUSDC could be valid input tokens for SY cUSDC where the USDC deposit would mint cUSDC on Compound). Ultimately, a 1:1 backing of yield tokens for the SY tokens is enforced. `previewDeposit()` and `previewRedeem()` both provide estimates for deposits and redemptions (intended for off-chain usage), respectively. Furthermore, each SY implements a `exchangeRate()` function that should return a value so that when an SY amount is multiplied with it, an accurate amount of underlying `asset` of the `yieldToken` is returned. Note that the exchange rate will be scaled by `10**18` and could require some action on the external system to refresh it.

Note that both implementations of the SY logic are abstract and that integration-specific behavior must be implemented in more derived contracts. However, it is expected that the exchange rate only depends on the external system.

Furthermore, Pendle Finance's SY implementation implements functions for collecting reward tokens (e.g. COMP tokens for cTokens). `claimRewards()` allows any user to claim for any address (to the address itself). `rewardIndexesCurrent()` and `rewardIndexesStored()` return respectively the current or a potentially outdated index for reward computations while `accruedRewards()` returns a potentially imprecise lower bound on the rewards accrued. `getRewardTokens()` is the set of reward tokens that are being received. A trivial version implemented for this is, is a version `SYBase` where the reward-specific functionality returns zeroes and does not do anything. However, `SYBaseWithRewards` is an implementation that can distribute rewards according to the mechanics of the abstract `RewardManager` contract. Note that the reward shares and the total reward shares used in said abstract contract will be the balances of users and the total supply of SY. Additionally, the rewards are accrued into the reward index and distributed among users before any balance changes. Thus, the invariants of the rewards manager should hold.

Last, the owner can `pause()` and `unpause()` an SY from being active.

2.2.2.1 Lido stETH SY

This contract implements a SY for stETH. Namely, it supports ETH, WETH, stETH and wstETH as input tokens where the two latter ones are valid output tokens. Note that wstETH is the yield-bearing token while stETH is the underlying asset. Hence, deposits would convert to wstETH so that 1 SY equals 1 wstETH.

2.2.3 Yield Tokenization: PY

Pendle Finance implements a system for yield tokenization that splits SYs into a principal token (PT) and a yield token (YT). The PT is a receipt token that allows holders to claim an amount of the SY's `yieldToken`'s underlying (called principal) that is equal to their PT balance. Hence, each PT represents a unit of the underlying asset in the external system that is generating yield. For each PT, a YT exists that gives a holder the right to claim the yield (both interest and reward tokens) generated by the principal of the PT. Thus, to ensure that for each YT a PT exists, they must be minted and burned together. However, to remove the requirement that PT and YT holders must hold the other token, an expiry date is set for the PT+YT (short: PY) mechanism. Post expiry, yield stops accruing so that no principal is needed for the YT's yield generation and, thus, the PTs and YTs can be redeemed independently.

More specifically, PTs and YTs can only be minted pre-expiry. They are minted according to the current exchange rate from SY to underlying asset (recall, that this is the exchange rate of the yield token that is wrapped by the SY). More formally, the amount of principal that a user provides in the form of SYs to the PY system is computed as

$$a_{PT} = a_{SY} r_{SY \to asset}$$

Given, that for each PT one YT must be minted $a_{PT} = a_{YT}$ holds.

`mintPY()` and `mintPYMulti()` implement the minting behavior described above. The latter is a batched version of the former that allows distributing the minted YTs and PTs to multiple receivers.

Redemptions work similarly to minting. Before expiry, both YT and PT need to be provided for redeeming the principal (see reasons above). Thus, $a_{PT} = a_{YT}$ holds too in that case. The amount of SY to be received for the principal by solving the above equation for a_{SY}. In case expiry has been reached, only PTs need to be redeemed. Thus, after expiry $a_{YT} = 0$ will hold. `redeemPY()` and `redeemPYMulti()` implement this mechanism for redeeming. The latter is a batched version of the former that allows distributing the SYs received to multiple receivers.

The PY system assumes that the exchange rate can only increase.

Hence, note that given the changing interest rate, the amount of retrievable SYs by the PT holders will decrease. The overhead of SYs is interpreted as interest. That interest is distributed fairly among the YT holders.

Rewards will be coming from the SY contract and are distributed among the users who have a claim to receive rewards. Meaning, that the rewards will be distributed to all YT holders and all addresses that have generated some SY as accrued amounts through interest (the overhead in underlying is generating rewards).

Recall that there is an expiry date after which YT holders do not receive any yield. The interest and the rewards will then be automatically forwarded to Pendle Finance's treasury and can be claimed to the treasury with `redeemInterestAndRewardsPostExpiryForTreasury()`. Regular users, however, can claim rewards and interest with the function `redeemDueInterestAndRewards()` at any time to receive their outstanding rewards.

Fees are paid out when interest and rewards are paid out to a user on `redeemDueInterestAndRewards()`. Meaning, if the rewards and interest accrued for a user so far is $reward_{user,\,token}$ and $interest_{user}$, respectively, the values $fee_{reward} reward_{user,\,token}$ and $fee_{interest} interest_{user}$ will be sent to the treasury. Thus, the reward mechanics can be "inherited" from the SY contract. Meaning, there is no need to track a global reward index since the SY contract provides it already. As a consequence, the `RewardManagerAbstract` reward mechanism is used where updating the global reward index (`_updateRewardIndex()`) is simply updating and getting the SY's global reward index. Since the user will accrue rewards with the principal from PTs and with the accrued SYs from interest, the reward shares balance (`_rewardSharesUser()`) of a user will be the share of SY's generating yield plus the accrued SYs from interest.

`pyIndexCurrent()` can update the cached PY index which is the exchange rate of SY to the underlying (note that `pyIndexStored()` only returns the cached value). Similarly, the `rewardIndexesCurrent()` function updates all reward indexes that are used to the most recent value.

The reward tokens can be queried with `getRewardTokens()` which essentially forwards the call to the SY contract.

Further, note that once expiry has been reached the state of the PY system at expiry must be recorded so that no yield can be received by YT holders after expiry. Hence, each action available after expiry will set the expiry state. Note that one can manually set the expiry date with `setExpiryData()`.

Last, a function to check whether the PY has matured is offered (`isExpired()`).

Note that the PY contracts can be deployed by the corresponding factory through the function `createYieldContract()` that deploys the YT contract, that implements the logic described above, and the PT contract that is a standard ERC20 contract where the YT can mint PTs.

2.2.4 Market

Pendle Finance implements a market for trading SY and PT which can be interpreted as a market for the SY's asset and the PT. As a consequence, PT holders can exit their PT prematurely. Given the time-to-maturity, the PTs will be traded at a discount so that the buyers of PTs will have a fixed interest, according to the buying price, that can be realized after the expiry. As a consequence, the exchange rate implies an interest rate. For example, assume that $exchangeRate$ is the price of the asset in terms of PT, then a user buying PTs at said price will profit at expiry. Namely, his implied interest will be equal to the exchange rate. However, when speaking of implied interest, we will refer to the interest per annum so that the interest can be defined as

$$impliedInterest = exchangeRate^{\frac{t_{year}}{t_{toExpiry}}}$$

Note that the Market should only allow swaps to change the implied interest rate. Namely, buying and selling PTs should decrease and increase the implied interest, respectively. Hence, between swaps, interest rate continuity should be enforced.

The exchange rate of the market is defined using the logit function over the proportion $p(t) = \frac{n_{PT}}{n_{PT} + n_{asset}}$. More specifically, the exchange rate will be

$$exchangeRate(t) = \frac{ln(\frac{p(t)}{1-p(t)})}{rateScalar(t)} + rateAnchor(t)$$

The rate scalar parameter effectively reduces the exchange rate and interest rate slippage the higher it becomes. Hence, the closer the system is to maturity, the rate scalar should increase. The rate scalar is defined as

$$rateScalar(t) = \frac{scalarRoot}{t}$$

Note that t is the normalized time to expiry over the PT's duration.

$rateAnchor(t)$ is a parameter to enforce the interest rate continuity property. Namely, the rate anchor will enforce that the definition of $impliedInterest$ still holds. Hence,

$$rateAnchor(t) = impliedInterest^{\frac{t_{toExpiry}}{t_{year}}} - \frac{\frac{p(t)}{1-p(t)}}{rateScalar(t)}$$

holds. Note that the definition allows for the interest rate to remain constant between swaps but will allow the exchange rate to change over time.

The swaps are implemented as an approximation. The swap exchange rate will be

$$exchangeRate_{trade} = \frac{\frac{p_trade}{1-p_trade}}{rateScalar(t)} + rateAnchor(t)$$

where

$$p_{trade} = \frac{n_{PT} - d_{PT}}{n_{asset} + n_{PT}}$$

Note that it can be shown that, as long as the exchange rate is above 1 (only meaningful exchange rates), the approximation does not lead to an arbitrageable scenario for a user swapping back and forth. Thus, it is enforced that the exchange rate is always at least 1.

Note that fees are taken as slippage on the interest rate. Namely, if a user buys PTs, the implied interest rate at which he trades should be less than it actually is so there is negative interest rate slippage for the user (buying less interest than implied). Similarly, if a user sells PTs, the implied interest rate at which he trades should be higher than it actually is so there is negative interest rate slippage for the user (selling more interest than implied). Consequently, the fee is defined as follows

$$impliedInterest_{trade} = impliedInterest * feeRateRoot$$

where the multiplication is done when a user sells PT.

As a consequence, the exchange rate is defined as

$$exchangeRate_{trade,\,fee} = exchangeRate_{a,\,trade} * feeRateRoot^{\frac{t_{toExpiry}}{t_{year}}}$$

The `swapExactPtForSy()` and the `swapSyForExactPt()` functions implement this trading mechanism. `mint()` is used for adding liquidity. To not change the implied interest when adding liquidity, the funds must be provided proportionally to the funds present in the contract. `burn()` is used for removing liquidity. Similarly, the funds are withdrawn proportionally so that the implied interest does not change. Note all these actions except for `burn()` are only allowed as long as expiry has not been reached.

Note that the Market does not pull funds but expects them to be pushed. With `skim()`, users can withdraw any overhead balance (e.g. the router may push too many funds and withdraw the overhead deposits).

Further, note that the market records the sum of $ln(impliedRate)$ (using the implied rates of the previous block), which can be retrieved with `observe()` (potentially weighted by time if no measurement at the queried timestamp exists). Ultimately, by using two such sums, a time-weighted average of the implied rate could be computed.

The market additionally implements a liquidity gauge where liquidity providers automatically stake their LP tokens to receive reward tokens which include the reward tokens received by the SY and the PENDLE token. The distribution of these rewards corresponds to the logic described in Reward Manager where the reward shares correspond to the so-called active balance. The active balance is defined as being the minimum between the LP balance a user holds, and the sum of 40% of his LP balance and his share, according to his vePENDLE share, of 60% of the LP supply. Through this mechanism, users can generate a 2.5x boost in their active balance. However, note that the active balance is only updated after user actions.

2.2.5 Oracles

Pendle Finance provides one PT oracle, an internal PT oracle library and an internal LP oracle library.

The PT oracle library implements

- `getPtToAssetRateRaw()` that returns the TWAP of the exchange rate or 1 if it is expired.
- `getPtToAssetRate()` returns the above but multiplies it with the ratio of the SY exchange rate and the PY index (SY index if it is outdated or the maximum of the stored PY index and the SY index, handling a potential PT depeg scenario). The two can be retrieved by the internal function `getSYandPYIndexCurrent()`.

The PT oracle's `getPtToAssetRate()` function wraps the corresponding function in the PT oracle library. `getOracleState()` is a helper function to determine whether an increase in cardinality is required for the desired TWAP duration, whether it can already be used for the TWAP duration, and what the required cardinality is for the TWAP duration. The required cardinality should be the number of blocks that could be at most present in the observations during the duration. Hence, if the next cardinality is not high enough, the observation cardinality must be grown. However, if the cardinality is sufficiently large,

then it must be ensured that the oldest observation would observe the duration properly. If that is the case, the oracle can be used.

The LP price oracle library offers `getLpToAssetRate()` to its users which computes the value of the LP token in terms of the asset according to the TWAP. Namely, it does so by using the asset and PT amount at the current exchange rate and reverse-engineering the asset and PT deltas of a hypothetical swap that moves the TWAP exchange rate to the current exchange rate. Given the reverse-engineered PT and asset state at the TWAP, the amounts are converted to assets and scaled by multiplying it with the ratio of the SY exchange rate and the PY index (SY index if it is outdated or the maximum of the stored PY index and the SY index, handling a potential PT depeg scenario).

2.2.6 Changes in Version 3

The market factory has been adapted. Namely, the following changes have been performed:

1. `vePendle` and `gaugeController` are now immutable.
2. There is no global `defaultLnFeeRateRoot` anymore. However, market creations take a `lnFeeRateRoot` argument. Hence, `markets` can share the same `PT`, `scalarRoot` and `initialAnchor` while having distinct `lnFeeRateRoot` values. Note that the `lnFeeRateRoot` of each market cannot exceed the maximum value.
3. Along with 2, `overriddenFee` is now more fine-grained and dependent on the router and on the market. It allows for overriding the `lnFeeRateRoot` of a router/market pair. Namely, that is done with `setOverriddenFee()`. Note that the value still cannot exceed the maximum value. `getMarketConfig()` now returns the `overriddenFee` for a router/market pair. If it is zero, the Market will use the `lnFeeRateRoot` set on creation.
4. `setTreasury()` and `setDefaultFee()` have been removed. However, `setTreasuryAndFeeReserve()` has been introduced and implements both functionalities. Note that constraints still hold.
5. Related refactoring (e.g. changes in events) has been introduced as a result.
6. The initializer has been removed and its setup has been moved to the constructor.

Note that the changes yielded changes in the market. `readState()` now handles the fees according to the description of 3.

2.2.7 Trust Model and Roles

Generally, it is expected that users are aware of how the system works and act accordingly (e.g. expected to know that transferring funds without a router to some contracts may lead to someone else using those funds, the markets have no slippage protection - expected to use routers). However, users are always untrusted.

Pendle Finance's contracts using the abstract reward manager are expected to not have any reward tokens that rebase down, have transfer fees or have unexpected behaviour (e.g. reverting when not expected to revert). Standard ERC-20s are expected. Additionally, it is expected that the reward tokens are not the tokens used by the contract inheriting from the reward manager (e.g. not the input/output/yield/... token for SY, not the SY for PY, or not the PT or SY for the market).

For the SY, the following trust model is defined along with some assumptions:

- The SY contracts have an owner that can pause the SY contract.
- The SY contracts' preview functions are only expected for off-chain use.
- The yield token that the SY wraps is expected to be an ERC-20 compatible token with a non-rebasing balance or transfer fees. Standard ERC-20 tokens are expected.
- The external system is expected to be fully trusted.
- The SY trust model fully depends on the more derived contracts that were out-of-scope.

The PY contracts have the following trust model, roles and assumptions:

- The trust model severely depends on the SY used and the external system as malicious SYs could for example lead to DOS attacks.
- The PY factory has an owner that can set the treasury, the fees (limited by 20%) and the expiry divisor. The latter could be used to only allow deployments of PY contracts with an expiry very far in the future.
- It is expected that the SYs used cannot have an exchange rate that decreases as this could lead to insolvency. Additionally, it is expected that SYs used are following the general patterns of the SY implementation provided (e.g. no rebasing SY, adhering to SY interface).

The market contract has the following trust model, roles and assumptions:

- The trust model severely depends on the SY used and the external system as malicious SYs could for example lead to DOS attacks.
- Similarly, it depends on the PY contracts used. Note that these are always expected to be Pendle Finance's yield contracts.
- The vePENDLE mechanism is out-of-scope and, hence, fully trusted as it could also allow for DOS attacks.
- The market factory is an ownable contract where the owner can set the treasury address and the default and custom router fees. While the fee itself is limited, the treasury's share of fees is not limited, see 100% reserve fee possible in market.

The owner in each contract is expected to act honestly.

Further, note that any usage of the library contracts outside of the usage in Pendle Finance's system is out of scope (e.g. usage of oracle library).

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	0
Low-Severity Findings	7

5.1 Frontrunning Initializers

Design Low Version 1 Code Partially Corrected Acknowledged

CS-PENDLEOCT2023-005

`PendleYieldContractFactory` and `PendleMarketFactory` have initializer functions. The calls to these functions may be frontrun.

Code partially corrected:

The issue for the market factory has been resolved.

Acknowledged

Pendle Finance acknowledged the risk of frontrunning and answered:

```
Noted on this; we won't fix it since it has already been deployed.
```

5.2 Inaccurate `userRewardOwed`

Correctness | Low | Version 1 | Acknowledged

CS-PENDLEOCT2023-007

In the PY system, `userRewardOwed` tracks the outstanding amount of rewards to be paid out to users post-expiry so that rewards received after expiry can be received by the treasury. However, that may be inaccurate due to donations. Thus, donated tokens may not be received by the treasury.

Acknowledged

Pendle Finance acknowledged this behavior.

5.3 Initial Liquidity Mismatches Whitepaper

Correctness | Low | Version 1 | Acknowledged

CS-PENDLEOCT2023-009

The initial liquidity is computed as the square root of `syDesired * ptDesired`. However, the whitepaper specifies it to be `d_asset`.

Acknowledged

Pendle Finance acknowledged the issue and answered:

```
The whitepaper is outdated here; it will be edited.
```

5.4 Mismatches With EIP-5115

Correctness | Low | Version 1 | Acknowledged

CS-PENDLEOCT2023-010

On several occasions, the SY implementation mismatches the version of EIP-5115 at the time of writing. Namely, the following mismatches are present:

1. `previewDeposit()` and `previewRedeem()` revert even though the standard states that they must be non-reverting.
2. The interfaces of `deposit()` mismatch due to a lack of the `depositFromInternalBalance` flag in the SY implementation.

Acknowledged:

Pendle Finance responded:

```
The EIP is outdated here; it will be edited.
```

5.5 Native Token Rewards Cannot Be Received by PY

Correctness Low Version 1 Acknowledged

CS-PENDLEOCT2023-011

The SY implementation indicates that ETH rewards could be received. Similarly, the PY contracts indicate proper handling of ETH rewards. However, these cannot be received due to a lack of a fallback function. Ultimately, having ETH as a reward could DOS the PY system.

Acknowledged

Pendle Finance acknowledged this behavior.

5.6 Rounding in Favour of User When Adding Liquidity

Correctness Low Version 1 Risk Accepted

CS-PENDLEOCT2023-014

When adding liquidity, the computation of the second used amount is rounded down. However, that is not in favour of the system. For example, when `netLpByPt < netLpBySy` holds, then the line

```
syUsed = (market.totalSy * lpToAccount) / market.totalLp;
```

will be rounding down. Note that the other branch creates the same problem.

Consider the following example,

1. Liquidity is initialized with 4000 PT and 251 SY so that liquidity will be the rounded down square root of `1004000` which is 1001 (minimum liquidity plus one).
2. Now, a user specifies `syDesired = 2**255` and `ptDesired = 4`. Hence the branch of the above line will be entered.
3. The `lpToAccount` will be `4*1001/4000=1`.
4. The above line compute `251*1/1000` and thus round down so that `syUsed` will be 0.

Ultimately, rounding is not done in favour of the system. Note that this can break the swaps as the proportion will exceed the maximum if done repetitively on low liquidity.

Risk accepted:

Pendle Finance accepts the risk:

```
Noted, we won't fix this for now since the market has already been deployed.
```

5.7 `msg.value` Can Be Non-Zero When `tokenIn` Not Being ETH

Design | Low | Version 1 | Risk Accepted

CS-PENDLEOCT2023-016

The SY contracts' `deposit()` function is payable to allow for native token deposits when `tokenIn` is set to the native token placeholder address. However, when the input token is another token, the native token can still be provided to the call without the code reverting. Thus, ETH could be stuck.

Risk accepted:

Pendle Finance replied the following:

> This is acceptable to us, as this behavior is applied to our routers as well.

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	0
High-Severity Findings	0
Medium-Severity Findings	2

- LP Oracle Library Manipulation Code Corrected
- LP Oracle Library Reverts After Expiry Code Corrected

Low-Severity Findings	2

- Incorrect Required Cardinality Code Corrected
- Not All Functionality Paused in SY Code Corrected

Informational Findings	1

- Gas Optimizations Code Corrected

6.1 LP Oracle Library Manipulation

Security Medium Version 1 Code Corrected

CS-PENDLEOCT2023-002

The library `PendlePtOracleLib` computes the rate from PT to Asset as the rate of PT to asset reported by the Market's oracle multiplied by the ratio `syIndex / pyIndex`. This additional multiplication is implemented to handle special cases where the underlying asset becomes insolvent and has a decreasing exchange rate.

In case the PY system caches pyIndex when queried multiple times during the same block (`YT.doCacheIndexSameBlock() == true`) it might be the case that the `syIndex` read by the Oracle library is greater than the `pyIndex`, leading to an inflated rate from PT to asset. Specifically, a malicious party could perform the following:

1. Call the PY system to cache the index for the current block.
2. Inflate the SY rate by performing a donation.
3. Still in the same block, call a protocol using the `PendlePtOracleLib`. The value returned by `getPtToAssetRate` will be inflated as the SY index used is larger than the cached PY index.

Code corrected:

The code has been adjusted to prevent the above attacks. Namely, if the SY index exceeds the PY index, no scaling will be performed. That prevents the overvaluation of PTs and LPs. However, note that protocols should carefully evaluate the suitability of the oracle contracts.

6.2 LP Oracle Library Reverts After Expiry

Security | Medium | Version 1 | Code Corrected

CS-PENDLEOCT2023-003

The LP oracle library's `_getLpToAssetRateRaw()` should return some rate even if expiry has been reached. However, `state.getMarketPreCompute()` will revert if expiry has been reached.

In the case of external protocols using this in their oracle contracts, the impact could be high due to unusable oracles.

Code corrected

`_getLpToAssetRateRaw()` now only calls `getMarketPreCompute()` if the expiry has not been reached and hence the function no longer reverts when called after expiry.

6.3 Incorrect Required Cardinality

Correctness | Low | Version 1 | Code Corrected

CS-PENDLEOCT2023-008

`PendlePtOracle._calcCardinalityRequiredRequired` computes the cardinality needed for TWAP over some duration. The current implementation however returns an incorrect computation of such cardinality as it is too small.

In the case of Ethereum for example, where `blockCycleNumerator == 11000` according to the documentation, The following durations would give the following cardinality required:

- 10s → `0`
- 11s → `1`
- 12s → `1`

while one would expect in each of those cases the cardinality required to be `2` (one entry for the block at timestamp `X-12` and one for the block at timestamp `X`).

Code corrected:

The issue has been addressed by rounding up the division and by increasing the resulting number of blocks by one. While for 12 seconds (for the above example) it may overestimate the blocks needed, it is ensured that a minimum required amount is always enforced.

6.4 Not All Functionality Paused in SY

Correctness | Low | Version 1 | Code Corrected

CS-PENDLEOCT2023-012

The pausing functionality should effectively pause all functions. However `claimRewards()` and `rewardIndexCurrent()` are not paused.

Code corrected:

The functions are now also being paused.

6.5 Gas Optimizations

Informational **Version 1** **Code Corrected**

CS-PENDLEOCT2023-017

1. In `RewardManager._updateRewardIndex()`, for each reward token, `rewardState[token].lastBalance` is read twice from storage.
2. In `RewardManager._updateRewardIndex()`, for each reward token whose index is being updated (`lastRewardBlock != block.number`), `rewardState[token].index` is read twice from storage.
3. `SYBaseWithRewards.rewardIndexesCurrent()` could already return the return value of `_updateRewardIndex()` instead of calling `rewardIndexesStored`.
4. In `PendleMarketFactory.getMarketConfig()`, in the case that `overriddenFee[router].active` is true, `defaultFee.lnFeeRateRoot` and `defaultFee.reserveFeePercent` are read from storage only to be overwritten.
5. In `PendleMarketFactory`, as `vePendle` and `gaugeController` are only set once in the function `initialize`, the two state-variables could be instead immutables that would be set in the constructor.
6. The `_emitNewMarketConfigEvent()` may lead to unnecessary storage loads.
7. `PT.YT` could be made an immutable. Precomputing the variable could allow to not use an initializer.

Code corrected

Most of the suggestions have been implemented (exception is 7.).

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 Incorrect NatSpec

Informational Version 1 Acknowledged

CS-PENDLEOCT2023-018

The NatSpec of several entry points of the system is missing or incomplete.

Acknowledged:

Pendle Finance acknowledges the issue.

7.2 Lack of Events

Informational Version 1 Acknowledged

CS-PENDLEOCT2023-019

The code lacks events on several occasions. For example, BoringOwnable lacks an event when setting pending owner.

Acknowledged

Pendle Finance acknowledged the lack of events.

7.3 Locked Rewards for `address(this)` in Reward Manager

Informational Version 1 Acknowledged

CS-PENDLEOCT2023-020

The abstract reward manager contracts do not update the reward index for `address(this)` and `address(0)`. That may lead to unclaimable rewards if either of the two has a reward share. For example, in the SY contracts, `address(this)` could technically hold SYs during a reward distribution. While that balance would be accounted for in the total reward shares, `address(this)` will never be able to claim the rewards nor update its index. Thus, the rewards could technically be distributed to other users. Similarly, that is the case in the yield contracts.

Acknowledged:

Pendle Finance acknowledges the issue.

7.4 Naming of Market LPs

Informational | Version 1 | Acknowledged

CS-PENDLEOCT2023-021

The naming of the LP tokens is always the same. However, a more meaningful name could help front-ends display the PT/SY market in a better way.

Acknowledged

Pendle Finance acknowledged that the naming of Market LP tokens could be more meaningful.

7.5 Unbound Array Size

Informational | Version 1 | Acknowledged

CS-PENDLEOCT2023-023

The system loads arrays from storage or external systems and then iterates over their items (e.g. reward tokens). These arrays could be large (or become large). Ultimately, computations could be DOSed if the array sizes are too high.

Acknowledged

Pendle Finance acknowledged the possibility of denial of service in the case the arrays become large.

7.6 Usage of Upgradeable Contract

Informational | Version 1 | Acknowledged

CS-PENDLEOCT2023-024

The system uses the `BoringOwnableUpgradeable` contract which is intended to be for upgradeable contracts. Thus, it reserves some storage slots. However, for example, SY contracts will not be upgradeable. Thus, this storage slot reservation could be unnecessary.

Acknowledged:

Pendle Finance acknowledged it.

7.7 `_stripSYPrefix()` Could Remove Too Much

Informational | Version 1 | Acknowledged

CS-PENDLEOCT2023-025

`_stripSYPrefix()` first removes the string "SY " (including the whitespace) and then "SY-" (no action if they are not present) assuming only one of the removal will be performed (the first one if the passed string is the token name, the second one if it is symbol name). However, if the SY is named "SY SY-..." for example, the algorithm of properly extracting the name will fail due to "SY SY-" being removed while only "SY " should have been removed.

Acknowledged

Pendle Finance noted the issue with `_stripSYPrefix()`.

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.

8.1 100% Reserve Fee Possible in Market

Note Version 1

LP providers should be aware that the reserve fee could be 100%, leading to no trading fees accrued for the LP providers.

8.2 Anyone Can Claim Rewards and Interests for a User

Note Version 1

Users should not assume that they are the only ones able to claim their claimable rewards or interest. Any user can claim for a user Alice so that Alice receives the rewards or interest.

8.3 Frontrunning-external System Updates

Note Version 1

Users should be aware the updates in external systems could be front-run. For example, if a user sees a transaction that will make rewards claimable for the SY system, he could frontrun that and receive free rewards as he holds a reward share prior to the reward update. Similarly, other reward or interest mechanisms could be front-run. However, typically that is not worthwhile to do due to rewards slowly increasing over time.

8.4 High Values for `expiryDivisor`

Note Version 1

Users can deploy the PY contracts using the factory. While users can deploy PY contracts at any time in the future, it is expected that they will only interact with ones that have a meaningful expiry (reachable in a reasonable amount of time). The factory enforces that the new expiry must be divisible by some expiry divisor to ensure that the expiry dates are consistent. If the `expiryDivisor` is chosen too high, no meaningful PY contracts can be deployed as their expiry would be too far in the future.

8.5 Hypothetical DOS of Swaps

Note Version 1

Users should be aware that swaps may not work under very extreme scenarios. However, adding and removing liquidity can be expected to work.

As an example of such extreme scenarios, consider the case where `(10**18-1)*totalPt + 1 = totalAsset` holds due to a rapid and extreme increase of the SY's exchange rate. Then, when computing the rate anchor for the swap, the current proportion will be computed to be `totalPt.divDown(totalPt + totalAsset)`. However, as we have defined `totalAsset`, we can replace it so that the formula becomes `totalPt.divDown(10**18*totalPt + 1)`. Note that the `divDown` only scales by `10**18`. Thus, the proportion will be `0` so that the ratio `0/(1-p)` is equal to 0. Taking the natural logarithm of `0` is undefined so that the swap is reverting.

8.6 Market Reward Distribution Can Be Unfair and Gamed

Note **Version 1**

In `PendleGauge`, when a user claims his rewards, the active balance used (`_rewardSharesUser(user)`) is the one that was set after the last LP balance change of the given user (`_afterTokenTransfer`).

Hence, as the vePendle balance of a user decreases naturally over time, for two users having the same vePendle and LP token balance, a user interacting often with the LP token will receive fewer rewards than a user that interacts less. Knowing that one can trigger an update of `_rewardSharesUser(user)` for another user by sending him 0 LP tokens for example, the reward distribution can be gamed where users update the reward shares of other users to increase their own rewards.

The following example shows an example where this behavior can lead to unfair reward distribution.

Let's have two users, Bob and Alice :

- they both have 100 LP tokens and a balance of 40 vePendle (same slope and bias)
- The total supply of LP is 200 (there is only Bob and Alice in the Market).
- The total supply of vePendle is 80 (only Bob and Alice have vested Pendle tokens).
- No rewards arrived yet, meaning that the global reward index is equal to 1.

According to `_updateUserActiveBalancePrivate`, after a transfer of 0 LP from Bob to Alice, their active Balance should be:

```
active_bal_alice = active_bal_bob = min(100, 100 * 0.4 + 200 * 40/80 * 0.6) =
 100
```

And hence, `totalActiveSupply = 200`

Now let's imagine that some time passes without any user action such that the vested Pendle token of both Alice and Bob has a balance of 2.

At this point, Charles, a user not in the market vests 300 Pendle tokens, hence having a balance of vePendle of 300.

Shortly after that, Bob transfer 0 LP token to Charles triggering first `_beforeTokenTransfer` for Bob, which does not do anything as `accrued = 0` but also `_afterTokenTransfer` which updates:

- Bob active balance to: `active_bal_Bob_1 = min(100, 100 * 0.4 + 200 * 2/(300 + 2 + 2) * 0.6) = 40.7`
- `totalActiveSupply` to `200 - 100 + 40.7 = 140.7`

Finally, 1000 rewards arrive and Alice does a 0 LP transfer to Bob, the new reward index is computed as `index = 1 + 1000/140.7 = 8.107`

This means that the following amount of reward is added respectively to Bob and Alice's accrued rewards:

- `accrued_alice = 100 * (8.107 - 1) ~= 710`
- `accrued_bob = 40.7 * (8.1 - 1) ~= 290`

Although Bob and Alice always had a matching balance of LP tokens and a matching balance of vePendle, Alice is getting more rewards than Bob.

8.7 Oracle Suitability

Note **Version 1**

Pendle Finance provides LP and PT oracle contracts. Users of these contracts (e.g. protocols integrating with these oracles), should carefully evaluate the usage of the oracle contracts and check their suitability for their specific project.

8.8 PT/YT Pre-Expiry Redemption

Note **Version 1**

Users should be aware that redeeming PT pre-expiry requires both PT and YT in the PY contract. The redeemed amount will be the minimum of the PY contract's YT and PT balances. If they are not equal, funds could be lost or taken by other users.

Similarly, in other parts of the system tokens have to be sent to the contracts. However, users should be aware that the consumption of said tokens should occur in the same transaction.

8.9 PY Assumes Exchange Rate of SY Only Increases

Note **Version 1**

The PY system assumes an exchange rate that is only increasing. Logic is implemented to prevent reverts in case of decreasing values. However, users should be aware of the consequences of such occurrences.

1. The PT will depeg. In case of a decrease, the minting and redemption will occur with a higher exchange rate. Namely, a SY is worth more PTs than it should be. Hence, minting gives too many PTs while redemption does not return enough SYs.
2. No interest will be collected as long as the highest exchange rate is not reached.

Ultimately, the system may break due to accounting issues. Users should be aware of that assumption when using the PY system.

8.10 Stuck Floating SY

Note **Version 1**

When minting the PY tokens, the floating amount of SY is used. In the case of `mintPYMulti()`, the sum of the amounts must be less than or equal to the floating amount. However, if the floating amount exceeds the to-be-used amount, the floating balance will be stuck, due to the `syReserve` being set to the current balance.

Ultimately, the full floating will be used to increase the `syReserve` while less could be used for minting operations.

8.11 Trade Approximation

Note **Version 1**

Note that the trade exchange rate only approximates an accurate trade. Thus, the user will spend more SY or receive less SY due to the approximation. However, users should be aware that doing multiple smaller trades may yield better results for them.

8.12 `accruedRewards()` Return Value

Note **Version 1**

The `accruedRewards()` function returns the cached `accrued` value for a user and does not try to compute a more recent value of it. Users should be aware, that after an interaction with the system, the value could become significantly higher.