Pendle Finance

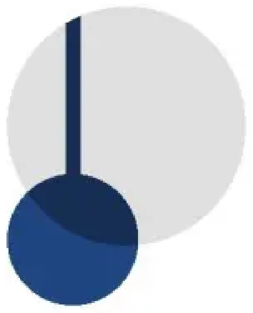
PendleV2 Part 1

Yield Tokenization & Trading

Audit

Smart Contract Security Assessment

July 1, 2022



ABSTRACT

Dedaub was commissioned to perform an audit on PendleV2 Part1 - Yield Tokenization and Trading - on branch `audit-dedaub` and commit hash a46985f1278c9c686d014446a8445d99a1b14991. The audited contract list is the following:

```
libraries/traderjoe/PendleJoeSwapHelperUpg.sol
libraries/SCY/SCYUtils.sol
libraries/SCY/SCYIndex.sol
libraries/helpers/MiniDeployer.sol
libraries/helpers/ArrayLib.sol
libraries/helpers/TokenHelper.sol
libraries/solmate/LibRLP.sol
libraries/math/LogExpMath.sol
libraries/math/MarketApproxLib.sol
libraries/math/Math.sol
libraries/math/MarketMathCore.sol
core/Market/PendleMarket.sol
core/Market/PendleMarketFactory.sol
core/actions/ActionCore.sol
core/actions/ActionYT.sol
core/actions/ActionCallback.sol
core/actions/base/ActionSCYAndPTBase.sol
core/actions/base/CallbackHelper.sol
core/actions/base/ActionSCYAndPYBase.sol
core/actions/base/ActionSCYAndYTBase.sol
core/PendleERC20.sol
core/YieldContracts/PendlePrincipalToken.sol
core/YieldContracts/PendleYieldContractFactory.sol
core/YieldContracts/PendleYieldToken.sol
core/PendleRouter.sol
SuperComposableYield/base-implementations/SCYBase.sol
SuperComposableYield/base-implementations/RewardManager.sol
SuperComposableYield/base-implementations/SCYBaseWithRewards.sol
SuperComposableYield/SCY-implementations/PendleBtrflySCY.sol
SuperComposableYield/SCY-implementations/PendleWstEthSCY.sol
```

```
SuperComposableYield/SCY-implementations/PendleERC4626SCY.sol
SuperComposableYield/SCY-implementations/PendleQiTokenSCY.sol
SuperComposableYield/SCY-implementations/PendleYearnVaultScy.sol
SuperComposableYield/SCY-implementations/AaveV3/PendleAaveV3SCY.sol
periphery/PermissionsV2Upg.sol
periphery/PendleGovernanceManager.sol
```

Two auditors worked on the codebase over three weeks. Our team's math consultant also contributed to this audit, more specifically in the PT AMM part of it.

Setting & Caveats

The audited codebase is of large size of ~5KLoC.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

The Pendle team provided us detailed documentation including whitepapers and other online documents, architectural diagrams and use case scenarios. The audited codebase is of excellent quality and of professional standards and is accompanied by an extensive test suite.

No Critical, High and Medium issues were identified which suggests a healthy protocol and implementation.

High-level Protocol Description

The first part of PendleV2 introduces a standalone smart contract system offering Yield Tokenization followed by Yield Trading.

The design of Yield Tokenization aims to cover the need for a unified interface of yield generating protocols in DeFi by introducing a new token standard, the Super Composable Yield (SCY). SCY is able to express the interaction with the majority of the yield generating mechanism in DeFi at the moment. The SCY standard essentially is an interface on top of the ERC20 token interface, including fundamental functions of yield generating protocols, such as `deposit()` and `redeem()`. The vision behind this Yield Tokenization standard is to be adopted as a universal yield token wrapper, so that more complex derivative systems, such as continuous strategy integrations in yield generating vaults, are easy to implement and extend considering a consistent interface. PendleV2 provides “base” implementation contracts for the SCY tokens, which can be extended when implementing a new, specific SCY token. These base implementations lie under the folder `SuperComposableYield/base-implementations`. The team has also implemented concrete SCY implementations for a number of yield protocols. These concrete implementations lie under the folder `SuperComposableYield/SCY-implementations`.

On top of the SCY standard the Super Composable Yield Stripping (SCYS) mechanism is built. SCYS is a mechanism to detach the yield- and principal-bearing aspect of an SCY token, by splitting it into two separate tokens; a Yield Token (YT) representing the yields and a Principal Token (PT) representing the principal value of the SCY token in respect to an expiry date. On top of the SCYS mechanism PendleV2’s Yield Trading part is built as an AMM for pairs of PT and SCY tokens. Pendle’s Principal Token AMM enables shorting or longing the fixed yield product PT. It is based on the AMM model for Notional but is more flexible regarding the AMM’s curve specifics allowing for increased capital efficiency.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming errors.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Yield token factory susceptible to time-bandit attacks	**RESOLVED**

In `PendleYieldContractFactory::createYieldContract`, a YT/PT contract pair is constructed with the classic CREATE mechanism:

```
function createYieldContract(address SCY, uint256 expiry)
    external
    returns (address PT, address YT)
{
    // [...]
    ISuperComposableYield _SCY = ISuperComposableYield(SCY);
    (, , uint8 assetDecimals) = _SCY.assetInfo();
    address predictedPTAddress = LibRLP.computeAddress(address(this),
++numContractDeployed);
    address predictedYTAddress = LibRLP.computeAddress(address(this),
++numContractDeployed);

    // Dedaub: Uses CREATE opcode under the hood
    PT = address(
        new PendlePrincipalToken(
            SCY,
            predictedYTAddress,
            PT_PREFIX.concat(_SCY.name(), expiry, " "),
            PT_PREFIX.concat(_SCY.symbol(), expiry, "-"),
            assetDecimals,
            expiry
        )
    );

    // Dedaub: Uses create opcode under the hood
    YT = _deployWithArgs(
             pendleYtCreationCodePointer,
             abi.encode(
             SCY,
             predictedPTAddress,
             YT_PREFIX.concat(_SCY.name(), expiry, " "),
             YT_PREFIX.concat(_SCY.symbol(), expiry, "-"),
             assetDecimals,
             expiry
             )
    );
    // [...]
}
```

However, as the newly created contract addresses only depend on the factory nonce, this can be exploited by a miner via time-bandit attack. More specifically, the miner could construct a malicious SCY token and call the factory create method with that parameter. With their own SCY token in place, they could manipulate the reported values in the YT and PT contracts. This way, if an opportunity arises where:

1. A user creates a YT/PT pair
2. Interacts with that pair in a separate transaction within the miners re-org window (e.g. by sending funds)

Then the miner could perform a time-bandit attack, create the YT/PT pair parametrized with his own malicious SCY token instead of the one intended by the victim and perform manipulations on the YT/PT pair.

Such an attack is only possible due to the fact that the constructor arguments are not taken into consideration during creation. This could be remedied with the use of a

CREATE2 approach, where the seed takes all YT/PT arguments into account. In this case, further engineering is required to resolve this, due to the cyclic dependency between the two tokens.

While this attack is definitely an edge case with a few strong assumptions, it is highly recommended that the code be fixed to be entirely protected against such attack scenarios.

L2	Unbounded “flash loan”-like logic in `PendleMarket::addLiquidity` could cause integration issues	**RESOLVED**

In `PendleMarket::addLiquidity`, the checks which verify that the required funds are present in the liquidity pool are done after the user callback returns, similar to how flash-loan/mint typically does:

```
function addLiquidity(
    // [...]
    if (data.length > 0) {
        IPMarketAddRemoveCallback(msg.sender).addLiquidityCallback(
             lpToAccount,
             scyUsed,
             ptUsed,
             data
        );
   }

    // have received enough SCY & PT
    require(market.totalPt.Uint() <= IERC20(PT).balanceOf(address(this)));
    require(market.totalScy.Uint() <= IERC20(SCY).balanceOf(address(this)));
    // [...]
}
```

The logic allows for an unbounded amount of liquidity tokens to be minted, before any commitment of underlying tokens has been performed, similar to a flash mint. While this is not an issue on its own, it can be problematic from a composability standpoint.

For example, other protocols that want to integrate with them could be exposed to overflow bugs that were thought to be impossible.

Similar concerns have also been raised about the flash mint functionality of other tokens, most notably `WETH10` which has seen a slow adoption rate for exactly this reason.

It is highly recommended that the code be refactored so that liquidity commitments are performed before the callback to prevent such issues.

It should be noted that similar logic appears in the `removeLiqudity` function. However, upon inspection, it was deemed to not be as impactful due to the fact that the amount burned is always limited by the depth of the liquidity pool.

L3	`Math::rawDivUp` does not revert when dividing by zero	**RESOLVED**

In `Math::rawDivUp` there is a check that early-returns 0 if the division is of the form 0/x:

```
// Dedaub: rawDivUp(0, 0) = 0, doesn't revert
function rawDivUp(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) return 0;
    else {
        return (a + b - 1) / b;
    }
}
```

However, this is not the expected behavior when the function is called as `rawDivUp(0, 0)`, in which case a revert should be performed.

While the code that uses the function always invokes it with a non-zero second argument, it is highly recommended to always revert in these cases to prevent any possible misuse in the future.

L4	Compiler issues	**RESOLVED**

Solidity compiler version v0.8.13 has, at the time of writing, some known bugs. We inspected the code and found that library SSTORE2.sol may be affected by the `InlineAssemblyMemorySideEffects` bug. We suggest either upgrading the compiler version or using the -via-IR pipeline when compiling the contracts.

OTHER/ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Incentivize community to call PendleMarket::skim	**DISMISSED**

Function PendleMarket::skim ensures handling the situation where a positive discrepancy between the actual token balance in the contract and the reserve number accounted for in the market appears. The excess amount of PT or SCY tokens are immediately transferred to the treasury.

Since this functionality guarantees the healthy state of the protocol it would be meaningful to incentivize the community's contribution by providing a percentage of the excess amounts as reward to the msg.sender.

ID	Description	STATUS
A2	Gas optimization: use local variable instead of storage field in emitted event	**RESOLVED**

In `PendleYieldContractFactory::setTreasury`, an event is emitted to reflect the change in the treasury field:

```
function setTreasury(address newTreasury) public onlyGovernance {
    require(newTreasury != address(0), "zero address");
    treasury = newTreasury;
    // Dedaub: Can be optimized by using newTreasury instead of treasury
    emit SetTreasury(treasury);
}
```

While the code is functionally sound, it can be optimized by emitting the provided parameter instead of the assigned storage field, saving some gas on each invocation.

A3	Code duplication	RESOLVED

Contract `PendleMarket` defines the constant variables

```
uint8 private constant _NOT_ENTERED = 1;
uint8 private constant _ENTERED = 2;
```

which are supposedly needed for the reentrancy guard, however these are already defined within the base contract `PendleERC20`.

A4	Potential issue prevented by coincidental check	INFO

The following is a near-issue we detected, that ended up not being realized due to a single, coincidental check.

We'll begin with the issue description first and then move on to how the current code does not allow this to happen:

The idea revolves around opening a YT/PT position for the YT token itself. The reason one wants to do that is to abuse the following fact: the address(YT) is exempt from the pre-transfer hook in the YT token, which is responsible for correctly updating the reward/interest state before the token transfer to avoid manipulation:

```
function _beforeTokenTransfer(
    address from,
    address to,
    uint256
) internal override {
    _updateRewardIndex();

    // Before the change in YT balance, users' impliedScyBalance is kept unchanged from
last time
    // Therefore, both updating due interest before or after due reward work the same.
    if (from != address(0) && from != address(this)) {
        _updateAndDistributeInterest(from);
        _distributeUserReward(from);
    }
    if (to != address(0) && to != address(this)) {
        _updateAndDistributeInterest(to);
        _distributeUserReward(to);
    }
}
```

With this in mind, an attacker could do the following:

1. Open a small YT/PT position on behalf of the YT token address.
2. Let it open for a long time
3. Repeat 1, with a much larger sum.
4. Trigger `updateAndDistributeInterest` for address(YT).

   Due to the pre-transfer hook exemption, address(YT) now appears as if it has been having a large YT/PT position open for the whole time since step 1, but this is clearly not the case, and then `redeemDueInterest`.

5. Call redeem with the appropriate values – the token assumes that the tokens must be held by the address(YT) at redeem time, which is true now due to step 4.

However, this issue does not manifest in the current code due to a simple check in the `PendleERC20::_transfer`, which requires that from and to must be different:

```
function _transfer(
    address from,
    address to,
    uint256 amount
) internal virtual {
    require(from != address(0), "ERC20: transfer from the zero address");
    require(to != address(0), "ERC20: transfer to the zero address");
    require(from != to, "ERC20: transfer to self");
    // [...]
}
```

This simple check prevents the realization of any “rewards”/interest that address(YT) would claim at the end of step 4, as `redeemDueInterest` would not be possible to call (eventually a SCY transfer from address(YT) to address(YT) is performed, and SCY tokens are PendleERC20 tokens).

As this check seems to address this issue by coincidence, a more concrete solution such as prevent self-mints would help prevent issues like this one.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.