Electisec

October 18, 2025

Prepared for
3Jane - Moneymarket

Audited by
Panda
Fede

3Jane - Moneymarket

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

3Jane is a credit-based money market built on top of Morpho Blue that provides unsecured credit lines underwritten against DeFi assets and FICO credit scores.

1.2 Audit Scope

This audit covers 13 smart contracts totaling approximately 2000 lines of code across 10 days of review.

```
src/
├── MorphoCredit.sol
├── Morpho.sol
├── ProtocolConfig.sol
├── CreditLine.sol
├── Helper.sol
├── MarkdownController.sol
├── InsuranceFund.sol
├── usd3/
│   ├── USD3.sol
│   └── sUSD3.sol
├── irm/
│   └── adaptive-curve-irm/
│       └── AdaptiveCurveIrm.sol
└── jane/
    ├── RewardsDistributor.sol
    ├── Jane.sol
    └── 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.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

Table 1: tab:severity-classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	2
Low	2
Informational	8

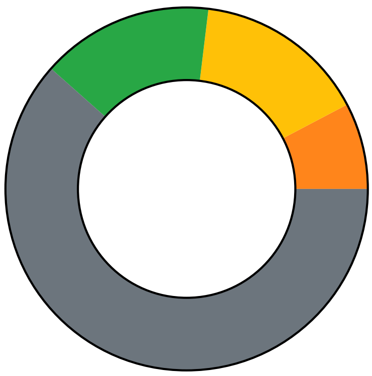

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The 3Jane Moneymarket protocol demonstrates solid architectural foundations with well-integrated external dependencies. The audit identified one high-severity issue in the Pendle YT token handling (subsequently removed from the codebase) and two medium-severity design issues around cooldown mechanics and token burn mechanisms that will be addressed in future releases. No critical vulnerabilities were discovered, and the core mathematical operations and access control systems are sound.

2 Audit Overview

2.1 Project Information

Protocol Name: 3Jane - Moneymarket
Repository: https://github.com/3jane-protocol/moneymarket-contracts/
Commit Hash: bcfbebfac56f0e8d1497efde6524eeb3e4d553ea
Commit URL: https://github.com/3jane-protocol/moneymarket-contracts/blob/bcfbebfac56f0e8d1497efde6524eeb3e4d553ea/

2.2 Audit Team

Panda, Fede

2.3 Audit Resources

Code repositories and documentation Whitepaper Previous audit

Category	Mark	Description
Access Control	Good	Access control mechanisms are generally well-implemented with proper role-based restrictions. Minor issues were found with whitelist validation in transfer hooks not being enforced when the whitelist is enabled, allowing potential bypass of access controls through transfers.
Mathematics	Good	Mathematical operations and calculations are sound with no critical vulnerabilities identified.
Complexity	Average	Several design issues stem from complex state management, such as the cooldown restart vulnerability and retroactive parameter modifications affecting existing users.
Libraries	Good	The protocol leverages well-established DeFi libraries and integrations, including Morpho Blue, and standard OpenZeppelin contracts.
Decentralization	Good	The system contains standard admin controls for parameter updates and contract management.
Code Stability	Good	Code stability is adequate with a mix of deployed immutable contracts and upgradeable components.
Documentation	Good	Documentation exists for core functionality.
Monitoring	Average	Event emission coverage is incomplete with several important state-changing functions lacking events.
Testing and verification	Average	Testing appears to cover basic functionality but missed several edge cases and mechanism vulnerabilities.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

2.5.1 Pendle `YT` tokens interests are lost during lock period

Technical Details

Pendle Yield Tokens (`YT`) are designed to accrue yield over time. When a user holds `YT` tokens, they receive yield payments through Pendle's mechanism (see docs).
However, `PYTLocker` contract locks `YT` tokens until their expiry but fails to handle the yield accrual mechanism. When users deposit `YT` tokens, the contract becomes the holder and receives the right to claim all yield payments, but these yields are never distributed back to the original depositors, resulting in permanent loss of yield for users.

Impact

High. Users lose all yield that would have accrued on their `YT` tokens during the lock period.

Recommendation

- Implement a yield claiming and distribution mechanism using a "rewards per token" pattern similar to staking contracts:

```
// SY tokens earned per YT token staked (scaled by 1e18)
mapping(address => uint256) public rewardPerTokenStored;

// User's reward per token checkpoint
mapping(address => mapping(address => uint256)) public userRewardPerTokenPaid;

// Unclaimed rewards for each user
mapping(address => mapping(address => uint256)) public rewards;
```

- Add a modifier when user interacts with the contract to claim from pendle and distribute.

```
modifier updateReward(address pytToken, address user) {
    // Claim all pending SY interest from Pendle YT contract
    (uint256 interestOut, ) = IPYieldToken(pytToken).redeemDueInterestAndRewards(
        address(this),
        true,   // redeemInterest
        false   // redeemRewards (if there are additional reward tokens)
    );

    // Update global reward per token
    if (totalSupply[pytToken] > 0) {
        rewardPerTokenStored[pytToken] += (interestOut * 1e18) / totalSupply[pytToken
];
    }

    // Update user's pending rewards
    if (user != address(0)) {
        rewards[pytToken][user] += balances[pytToken][user]
```

```
                    * (rewardPerTokenStored[pytToken] - userRewardPerTokenPaid[pytToken][user
])
                    / 1e18;
                userRewardPerTokenPaid[pytToken][user] = rewardPerTokenStored[pytToken];
            }
            _;
        }
```

- Apply the modifier to deposit and withdraw functions.
- Add the user claim mechanism with the corresponding SY token.

Note: This suggestion considers that no additional rewards are added to the interests.

Developer Response

Removed this functionality for now PR#89

2.6 Medium Findings

2.6.1 Cooldown restart allows users to bypass cooldown mechanism

Users can repeatedly call `cancelCooldown()` and `startCooldown()` to reset their cooldown timer while maintaining shares in an active cooldown state. This allows them to keep shares ”ready for withdrawal” without any opportunity cost, effectively bypassing the intended cooldown period protection.

Technical Details

1. **No restriction on restarting cooldown**: Users can call `startCooldown()` multiple times, with each call overwriting the previous cooldown state and resetting `cooldownEnd` to `block.timestamp + cooldownPeriod`.
2. **Shares continue earning yield**: Shares placed in cooldown remain as normal sUSD3 shares and continue earning yield from the USD3 strategy. There is no penalty or opportunity cost for having shares in cooldown.
3. **Strategic timing advantage**: A user can repeatedly call `startCooldown()` every few days to maintain a rolling cooldown window. When they actually want to withdraw, they only need to wait from their most recent `startCooldown()` call.

Impact

Medium. By allowing cooldown restarts, users can maintain ”withdrawal readiness” at all times without opportunity cost.

Recommendation

Implement a snapshot mechanism where shares in cooldown don’t earn new yield but are still exposed to losses:
Key principle:

- If share price **increases** during cooldown → user only gets the snapshotted value (no yield gains)

- If share price **decreases** during cooldown → user is affected by losses (first-loss protection still works)

1. **Update the `UserCooldown` struct** to include snapshotted assets:

```
struct UserCooldown {
    uint64 cooldownEnd;        // When cooldown expires
    uint64 windowEnd;          // When withdrawal window closes
    uint128 shares;            // Shares locked for withdrawal
    uint256 snapshotAssets;    // Asset value when cooldown started (NEW)
}
```

2. **Modify `startCooldown()`** to snapshot the current value.
3. **Update `availableWithdrawLimit()`** to use minimum of snapshot and current value.
4. Update `cancelCooldown` to burn shares to maintain the same number of underlying.

Notes: When a user withdraws with the snapshot mechanism and the share price has increased:

1. User receives `snapshotAssets` (lower than current value)
2. But `shares` are burned from the total supply
3. The difference between the current share value and the snapshotted value remains in the contract
4. This immediately increases the price per share for remaining users

Developer Response

ACK, but will not fix in the current release. We will fix in a subsequent release.

2.6.2 `JANE` burn mechanism is unfair and gameable

The `JANE` burn mechanism has some flaws that lead to an unfair and gameable process.

Technical Details

The burn mechanism relies on a snapshot of the borrower's balance taken only at the time of the first burn in `MarkdownController.burnJaneProportional()`. Any `JANE` received after the snapshot is not incorporated into the target burn, causing systematic under-penalization.
`MarkdownController.burnJaneProportional()` and `MarkdownController.burnJaneFull()` relies on the transfer freeze mechanism to prevent borrowers from moving their `JANE` tokens during default. However, once transfers are globally enabled, borrowers can transfer their `JANE` tokens to other addresses before entering delinquent/default status, effectively avoiding the penalty mechanism.
The penalty does not consider debt magnitude. Two borrowers with equal `JANE` balances but very different outstanding debts accrue the same burn curve, which is not proportional to credit risk contribution.

Impact

Medium. The burn mechanism is not fair and can be gamed.

Recommendation

If the intention is to prevent bad actors farming jane and then defaulting, consider to implement one of the following:

- Replace liquid `JANE` emissions with a non-transferable, vesting reward token (e.g. `veJANE`) and have the penalty mechanism burn unvested `veJANE`. Upon entering delinquent/default status, immediately stop vesting and farming. Take a deterministic snapshot at the state transition, and optionally scale the penalty by outstanding debt for better fairness.
- Allow burning not yet claimed `JANE` with the function that helps burn unclaimed `JANE` from the RewardsDistributor using a Merkle proof.

Developer Response

ACK, but will not fix immediately. In the short run, JANE will be non-transferable. We will come up with a better mechanism in the longer term.

2.7 Low Findings

2.7.1 USD3 transfer checks ignore whitelist in `_preTransferHook`

Technical Details

The `_preTransferHook()` function in USD3 enforces commitment period restrictions but does not validate whitelist requirements when `whitelistEnabled` is true. This allows whitelisted users to transfer their USD3 shares to non-whitelisted addresses, effectively bypassing the whitelist access control.

Impact

Low

Recommendation

Add whitelist validation to `_preTransferHook()` when whitelist is enabled.

Developer Response

Acknowledged, will not fix. Whitelist will be disabled soon, and whitelisting can be removed in a future release.

2.7.2 Commitment period can be retroactively modified

Technical Details

The USD3 contract enforces a minimum commitment period to prevent users from withdrawing immediately after depositing. However, the implementation stores only the `depositTimestamp` and dynamically calculates the commitment end time by reading `minCommitmentTime()` from the ProtocolConfig. Since `minCommitmentTime()` reads from ProtocolConfig, any changes to this parameter will retroactively affect all existing depositors.

Impact

Low.

Recommendation

Store the commitment end timestamp directly instead of recalculating it dynamically.

Developer Response

Acknowledge.

2.8 Gas Savings Findings

None.

2.9 Informational Findings

2.9.1 Update misleading comment about subordination ratio enforcement

Technical Details

NatSpec comment in sUSD3.availableDepositLimit() states the subordination ratio is enforced relative to `USD3` total supply, but the implementation uses market debt as the base.

Impact

Informational.

Recommendation

Update comments to reflect debt-based subordination enforcement.

Developer Response

fixed PR#88

2.9.2 Code duplicate

Technical Details

A function already exists to wrap USDC into WASUSDC; it's used as part of repay, but not for full replay.

```
File: Helper.sol
144:         IERC20(USDC).safeTransferFrom(msg.sender, address(this), usdcNeeded);
145:         IERC4626(WAUSDC).deposit(usdcNeeded, address(this));
// Can be replaced by
144:         _wrap(msg.sender, usdcNeeded);
```

Helper.sol#L144-L145

Impact

Informational

Recommendation

Update the code to remove duplication.

Developer Response

fixed PR#88

2.9.3 Cap `cover` to borrower's total debt instead of `assets` in `CreditLine.settle()`

Technical Details

`CreditLine.settle()` accepts an `assets` parameter that should represent the assets to settle. However, the function always settles the full position and optionally repays using `cover`, which is passed directly without capping it to `assets` while it should be capped to the borrower's actual outstanding debt.

Impact

Informational.

Recommendation

Compute the borrower's current debt and cap `cover` to that amount. Remove `assets` from the signature to avoid ambiguity.

Developer Response

Acknowledged, but will not update. `CreditLine` is already deployed and non-upgradeable. This doesn't rise to the level of an issue that warrants redeployment.

2.9.4 Missing event emission

Technical Details

The following setters are missing events:

- CreditLine setOzd(), setMm(), setProver(), setInsuranceFund()
- MorphoCredit.sol setHelper(), setUsd3()

Impact

Informational

Recommendation

Consider if an event is missing and add it if needed.

Developer Response

Ack, will not fix. We don't have bytecode headroom in MorphoCredit and CreditLine is immutable and already deployed

2.9.5 Unused import

The identifier is imported but never used within the file.

Technical Details

```
File: src/CreditLine.sol

8: import {EventsLib} from "./libraries/EventsLib.sol";
```

CreditLine.sol#L8

```
File: src/MorphoCredit.sol

18: import {IMorphoRepayCallback} from "./interfaces/IMorphoCallbacks.sol";

27: import {MathLib, WAD} from "./libraries/MathLib.sol"; // (WAD)
```

MorphoCredit.sol#L18, MorphoCredit.sol#L27

```
File: src/irm/adaptive-curve-irm/AdaptiveCurveIrm.sol

11: import {ConstantsLib} from "./libraries/ConstantsLib.sol";

16: import {IAaveMarket, ReserveDataLegacy} from "./interfaces/IAaveMarket.sol"; // (
ReserveDataLegacy)
```

AdaptiveCurveIrm.sol#L11, AdaptiveCurveIrm.sol#L16

```
File: src/usd3/sUSD3.sol

4: import {
5:     BaseHooksUpgradeable, IERC20, IMorphoCredit, IProtocolConfig, IStrategy, Math,
SafeERC20, USD3
6: } from "./USD3.sol"; // (SafeERC20)
```

src/usd3/sUSD3.sol#L4

Impact

Informational

Recommendation

Remove unused import to improve code quality

Developer Response

fixed PR#88

2.9.6 Unnecessary cast

The variable is being cast to its own type

Technical Details

```
File: src/CreditLine.sol

218: IERC20(marketParams.loanToken).approve(address(MORPHO), cover);
```

CreditLine.sol#L218

```
File: src/irm/adaptive-curve-irm/AdaptiveCurveIrm.sol

189: return (coeff.wMulToZero(err) + WAD).wMulToZero(int256(_rateAtTarget));
```

AdaptiveCurveIrm.sol#L189

```
File: src/usd3/USD3.sol

713: uint256 newSlotValue = (currentSlotValue & mask) | (uint256(trancheShare) << 32);
```

USD3.sol#L713

Impact

Informational

Recommendation

Simplify the code by removing the unnecessary cast.

Developer Response

fixed PR#88

2.9.7 `RewardsDistributor.getClaimable()` ignores global cap

Technical Details

`getClaimable()` returns the user's uncapped delta `totalAllocation - claimed[user]` but ignores the global cap based on `maxClaimable - totalClaimed`. The cap is only enforced during `_claim()`, where the amount is reduced to the remaining global cap.

Impact

Informational.

Recommendation

Change `getClaimable()` to apply the cap:

```
function getClaimable(address user, uint256 totalAllocation) external view returns (
uint256) {
    if (maxClaimable == 0 || totalClaimed >= maxClaimable) return 0;

    uint256 alreadyClaimed = claimed[user];
    uint256 uncapped = totalAllocation > alreadyClaimed ? totalAllocation -
alreadyClaimed : 0;

    uint256 remaining = maxClaimable - totalClaimed;
    return uncapped > remaining ? remaining : uncapped;
}
```

Developer Response

Acknowledged.

2.9.8 `RewardsDistributor.Claimed` event emits incorrect user total claimed

Technical Details

`Claimed` event docs state the third parameter is:

```
/// @param totalClaimed The total amount the user has claimed after this claim
event Claimed(address indexed user, uint256 amount, uint256 totalClaimed);
```

However, the emission passes `totalAllocation` instead of the updated user total claimed.

Impact

Informational.

Recommendation

Emit the updated cumulative user total `claimed[user]` after state updates:

```
emit Claimed(user, claimable, alreadyClaimed + claimable);
```

Developer Response

Addressed in PR#86