yAudit

May 2026

Prepared for
3Jane Money Market

Audited by
Panda
Fedebianlu

3Jane Money Market Contracts - USD3/sUSD3 Update

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

3Jane Money Market extends Morpho Blue with credit-line lending and the USD3/sUSD3 senior-junior strategy system.

1.2 Audit Scope

The review covered the production Solidity code changes introduced by PR #116.

```
src/
  ├── libraries
  │   └── ProtocolConfigLib.sol
  └── usd3
      ├── USD3.sol
      ├── sUSD3.sol
      ├── base
      │   └── BaseHooksUpgradeable.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	3
Medium	5
Low	1
Informational	0

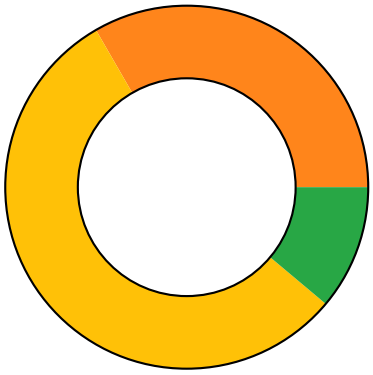

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The review identified 9 findings: 3 High, 5 Medium, and 1 Low. The main risks were stale accounting, report ordering, and USD3/sUSD3 liquidity controls.

2 Audit Overview

2.1 Project Information

Protocol Name: 3Jane Money Market
Repository: https://github.com/3jane-protocol/moneymarket-contracts
Commit Hash: b593f05f7159ecf9be257951077c680c14512612
Final Commit Hash: 52bf5322f53ed2e59fa31c994f76d4253d9b9172
Commit URL: https://github.com/3jane-protocol/moneymarket-contracts/commit/b593f05f7159ecf9be257951077c680c14512612

2.2 Audit Timeline

The audit was conducted from May 26 to 28, 2026.

2.3 Audit Resources

- Pull request #116 and linked USD3/sUSD3 update pull requests

Category	Mark	Description
Access Control	Good	Privileged roles and keeper/management boundaries are generally explicit. The reviewed issues were not primarily access-control failures, although several flows depend on trusted keepers and operational sequencing.
Complexity	Low	USD3 combines MorphoCredit accounting, waUSDC wrapping, Yearn TokenizedStrategy hooks, and sUSD3 first-loss behavior. The interaction complexity is high and contributed directly to several findings.

Table 2: Code Evaluation Matrix

2.4 Critical Findings

None.

2.5 High Findings

2.5.1 sUSD3 withdrawals create an over-cap borrow window that can strand senior liquidity

Technical Details

The `sUSD3.availableWithdrawLimit()` function enforces a backing floor based on `getSubordinatedDebtFloorInUSDC()`, which calculates the required junior backing using only the currently borrowed debt from the Morpho market. This creates a disconnect between the withdrawal floor and the actual exposure USD3 holds in the market.
When an sUSD3 holder completes their cooldown and withdraws, the backing floor is computed from the current debt at that moment. Because debt is typically lower than total supplied liquidity, the floor permits withdrawal of most junior capital. For example, if 10M USDC is supplied but only 1M is borrowed with a 15% backing ratio, the floor requires only 150k backing to remain.
The withdrawal flow in `_postWithdrawHook()` updates only cooldown state and does not trigger any synchronous deleveraging of USD3's Morpho position. Meanwhile, `_subordinationDeployCapWaUSDC()` immediately recalculates USD3's allowed deployment based on the reduced sUSD3 backing, dropping the cap from ~10M to ~1M in the example above. However, USD3 only enforces this new cap during later maintenance operations via `_tend()` or `_harvestAndReport()`.
This timing gap creates a vulnerable window: the 9M excess supply remains in Morpho and is still borrowable by addresses with valid credit lines. If a borrower draws this liquidity before USD3 deleverages, `_withdrawFromMorpho()` cannot reclaim the funds because it is constrained by `waUSDCLiquidity`, which becomes exhausted once the excess supply converts to debt. The system then remains in a state where debt materially exceeds the live subordination cap and cannot be corrected through tending until borrowers repay.
The root cause is that the sUSD3 withdrawal path enforces a floor against actual debt without ensuring that still-supplied and borrowable liquidity is first reclaimed. The protocol's subordination guarantee fails at the moment new debt is created because junior capital has already exited while senior liquidity remains exposed in the market.

Impact

High. A cooled sUSD3 holder can withdraw most of their position when debt is low, immediately reducing USD3's subordination-based deployment cap. An approved borrower can then draw the excess Morpho liquidity that should have been pulled back but remains borrowable. This leaves USD3 persistently above its live subordination cap, creates under-backed credit exposure, and strands most senior liquidity until borrowers repay or losses are resolved.
Senior tranche holders lose access to their capital except for a small local buffer that matched the pre-attack debt level. The first-loss protection buffer is materially smaller than intended at the exact moment new debt is created. Keepers cannot restore the cap through `tend()` because `_withdrawFromMorpho()` can only reclaim funds up to then-current market liquidity, which has been consumed.

The practical blast radius is bounded by the smaller of the over-cap supplied liquidity, unused borrower credit-line headroom, and protocol debt-cap headroom. The path requires realistic but non-trivial conditions: a matured sUSD3 position, an active borrowing cycle, and an address with a valid credit line. These are normal protocol states rather than privileged shortcuts.

Recommendation

Make junior-backing reductions and USD3 deleveraging atomic. When sUSD3 processes a withdraw or redeem, compute the post-withdrawal allowed USD3 deployment, force USD3 to withdraw Morpho supply down to that cap in the same transaction, and revert if enough liquidity cannot be reclaimed.
A practical implementation would add a USD3 entrypoint callable by sUSD3 that accepts the new backing level, calculates the corresponding deployment cap, and attempts to reclaim any excess supply before the sUSD3 withdrawal completes. This ensures that junior capital cannot exit while still-borrowable senior liquidity remains in the market.
If atomic deleveraging proves impractical, conservatively block sUSD3 withdrawals using the greater of the current debt-backed floor, any nominal floor, and still-supplied borrowable exposure until deleveraging has completed. This prevents the vulnerable window from opening but may reduce sUSD3 liquidity during rebalancing periods.

Developer Response

Fixed in bfa8c39.

2.5.2 Unreported realized losses let early USD3 withdrawers exit at a stale share price

Technical Details

USD3 uses live liquidity for withdrawal capacity, but stale Yearn accounting for share pricing. When `MorphoCredit.settleAccount()` writes off a defaulted borrower, it immediately reduces the Morpho market's `totalBorrowAssets` and `totalSupplyAssets`. This makes the loss real in MorphoCredit state and reduces the amount that USD3 can recover from the market.
However, USD3's `TokenizedStrategy.totalAssets()` is only updated when a keeper calls `USD3.report()`. Until that report runs, Yearn's `redeem()` and `withdraw()` paths still price USD3 shares using the old pre-loss `totalAssets`.
The relevant flow is:

- `src/MorphoCredit.sol:834` settles a defaulted borrower.
- `src/MorphoCredit.sol:874` clears the borrower position and applies the write-off.
- `src/MorphoCredit.sol:889` to `src/MorphoCredit.sol:897` reduces market `totalSupplyAssets` by the realized loss.
- `src/usd3/USD3.sol:442` computes `availableWithdrawLimit()` from live Morpho/waUSDC liquidity.
- `lib/tokenized-strategy/src/TokenizedStrategy.sol:615` prices `redeem()` using stored strategy accounting.
- `lib/tokenized-strategy/src/TokenizedStrategy.sol:920` converts the live asset withdrawal limit back into shares using stale `totalAssets`.

- `lib/tokenized-strategy/src/TokenizedStrategy.sol:989` burns shares and transfers assets according to that stale share price if the assets can be freed.

This creates a window after settlement and before USD3 report where early redeemers can exit at the old share price even though the loss has already been realized in MorphoCredit.

Impact

High. Early USD3 withdrawers can receive more than their post-loss pro-rata share, pushing extra loss onto remaining USD3 holders.

Recommendation

Before servicing USD3 withdrawals or redemptions, ensure share pricing cannot use stored `totalAssets` when live recoverable assets are lower due to a realized MorphoCredit loss.

Developer Response

Fixed in 8ab2d6c.
The patch adds a live NAV guard to `USD3.availableWithdrawLimit()`: if `USD3.nav() + 2 < TokenizedStrategy.totalAssets()`, `availableWithdrawLimit`, `maxWithdraw`, and `maxRedeem` all return zero until `USD3.report()` refreshes cached Yearn accounting. The guard is intentionally enforced before the shutdown bypass so shutdown cannot reopen stale-priced exits.

2.5.3 Loss reports can finish with borrowable liquidity above the post-burn backing cap

Technical Details

PR #110 describes the new subordination deployment cap as a rebalance-point invariant. However, `USD3.report()` can itself finish above the new cap after a loss because the report-time tend runs before sUSD3 loss absorption.
The relevant order is:

- `src/usd3/USD3.sol:563` captures pre-report totals and calls `_reportInternal()`.
- Inside the Yearn report flow, `src/usd3/USD3.sol:335` runs `_harvestAndReport()`.
- `src/usd3/USD3.sol:340` calls `_tend()` before the loss is recorded and before sUSD3 shares are burned.
- `src/usd3/USD3.sol:573` burns sUSD3's USD3 shares after `_reportInternal()` returns.
- No second tend or pullback runs after that burn, so `suppliedWaUSDC()` can remain above `effectiveDeployCapWaUSDC()`.

A focused integration test was temporarily added and removed during validation. It showed this sequence succeeds:

1. Deposit 1,000,000 USDC into USD3 and move 500,000 USD3 into sUSD3.
2. Set `MIN_SUSD3_BACKING_RATIO = 100%` and tend; USD3 deploys about 500,000 waUSDC.
3. Simulate a 400,000 waUSDC loss from the local buffer.

4. Call `USD3.report()`.
5. The report burns sUSD3 backing down to about 100,000 USD3, but supplied waUSDC remains about 500,000.
6. A credit-lined borrower borrows the roughly 400,000 waUSDC excess before the next tend.

Impact

High.

Recommendation

Ensure the sUSD3-backed deployment invariant is restored before `USD3.report()` returns after loss absorption. The implementation should account for the fact that the effective cap can decrease when `_postReportHook()` burns sUSD3-held USD3 shares.
The exact implementation can be chosen by the team, but `report()` should not be able to finish with borrowable MorphoCredit liquidity above the post-loss sUSD3-backed cap. This can be addressed within the USD3 report/loss-absorption flow; a broader borrow-time guard would be defense-in-depth rather than required to resolve this report-created violation.

Developer Response

Fixed in 24c4642.

2.6 Medium Findings

2.6.1 USD3 deposits can capture borrower premiums earned before entry at a stale share price

Technical Details

USD3 values its position in MorphoCredit using `expectedSupplyAssets()`, which relies on `expectedMarketBalances()` in `src/libraries/periphery/MorphoBalancesLib.sol:33-57`. This function models only base IRM interest accrual and does not include borrower-specific premium or penalty amounts that accumulate between explicit borrower touches.
In MorphoCredit, borrowers can carry premium or penalty rates that accrue value over time, but this accrued value remains unmaterialized until someone calls `accruePremiumsForBorrowers()` in `src/MorphoCredit.sol:121-136`. When that function runs, `_accrueBorrowerPremium()` calculates the accumulated premium and adds it to the market's `totalSupplyAssets` by issuing new borrow shares to the borrower. Because `accruePremiumsForBorrowers()` is public and permissionless, an attacker can control when this materialization occurs.
USD3's `_harvestAndReport()` in `src/usd3/USD3.sol:335-345` calls `accrueInterest()` to synchronize base IRM state, then values the Morpho position via `suppliedWaUSDC()` at line 684, which reads `expectedSupplyAssets()`. If borrowers have accumulated premium since their last touch, that premium is economically owed to current USD3 holders but is absent from the reported position value. A new depositor can therefore mint shares at this understated price, then immediately trigger the public premium-accrual function to materialize the missing yield. The next keeper `report()` recognizes that newly materialized premium as profit, which is then distributed across the enlarged share supply, allowing the attacker to capture a portion of yield earned before their deposit.

This creates a concrete exploit path: the attacker deposits into USD3 at a stale price-per-share that excludes accrued borrower premiums, calls `accruePremiumsForBorrowers()` to synchronize those premiums into the market state, and benefits when the next report distributes that historical premium across all current shares, including the attacker's freshly minted shares.

Impact

Medium. Existing USD3 holders lose a portion of borrower premium that accrued during their holding period when a new depositor enters before that premium is materialized. The amount lost is bounded by the outstanding premium backlog rather than principal, but it represents a real value transfer. The attacker receives more shares than a premium-inclusive price would allow, then exits with additional assets funded by yield earned before their deposit.
The PoC demonstrates this dilution with a 365-day premium accumulation period. After the attacker deposits and triggers premium synchronization, approximately 204.8 billion units of stale premium are materialized. The attacker receives 100 billion shares instead of the fair 86.6 billion shares and ultimately extracts about 18.9 billion extra asset units attributable to the stale-premium mispricing.
This also distorts the intended USD3/sUSD3 profit split because sUSD3 receives a share of reported profit. When historical premium is recognized after new share issuance, the distribution calculation includes shares that were not present when the premium was earned.

Recommendation

USD3 should not issue shares or finalize reports from a MorphoCredit state that omits accrued borrower premiums. The safest fix is to maintain a canonical set of active borrowers and fully accrue all borrowers before deposit pricing and `report()` execution. This can be implemented by calling `accruePremiumsForBorrowers()` with the complete borrower list as a required precondition in both the deposit flow and `_harvestAndReport()`.
If maintaining a full borrower set on-chain is not feasible, USD3 should replace the current valuation path with premium-aware accounting that includes all pending borrower premium before shares are issued. Alternatively, deposits could be conservatively restricted whenever borrower premium state may be stale, though this approach may degrade user experience. The goal is to ensure that new depositors cannot mint shares at a price that excludes value already earned by existing holders.

Developer Response

If profit unlocking is 7 days, the damage from this is somewhat limited. This is something we try to handle operationally by regularly accruing premiums for all active borrowers and using profit unlock periods of >=7d

2.6.2 Unsynced sUSD3 accounting lets junior holders redeem at stale pre-loss pricing

Technical Details

The sUSD3 vault is a Yearn-style ERC4626 strategy that wraps USD3 tokens. Its share pricing relies on TokenizedStrategy's cached `totalAssets`, which is updated only when the keeper calls `sUSD3.report()`. The cached value is returned by `_harvestAndReport()` at

`src/usd3/sUSD3.sol:119-125`, which simply reads `asset.balanceOf(address(this))` at report time.
The USD3 contract can directly mutate sUSD3's USD3 balance outside of sUSD3's own reporting cycle. When USD3 realizes a loss, `USD3._postReportHook()` at `src/usd3/USD3.sol:573-599` invokes `_burnSharesFromSusd3()` at `src/usd3/USD3.sol:640-667`, which directly decrements sUSD3's USD3 share balance and total supply via storage manipulation. This burn occurs atomically within `USD3.report()` but does not trigger any update to sUSD3's cached `totalAssets`.
User redemption flows in `BaseHooksUpgradeable` at `src/usd3/base/BaseHooksUpgradeable.sol:65-122` delegate directly to TokenizedStrategy's `redeem()` and `withdraw()` functions without first checking whether the cached accounting is stale. TokenizedStrategy then prices the redemption using the cached `totalAssets`, which may no longer match the live USD3 balance held by sUSD3.
This creates an exploitable window: after `USD3.report()` burns USD3 from sUSD3 but before a separate keeper calls `sUSD3.report()`, an attacker can redeem sUSD3 shares at the stale, higher `totalAssets` value and receive more USD3 than their post-loss entitlement. The attacker must have a matured cooldown or operate during shutdown when cooldowns are bypassed, as enforced by `availableWithdrawLimit()` at `src/usd3/sUSD3.sol:250-338`.
The stale-pricing window exists because the contracts do not enforce atomic batching of `USD3.report()` and `sUSD3.report()`, nor do user entry and exit paths in sUSD3 synchronize cached accounting before executing pricing logic.

Impact

Medium. After a USD3 loss event, a prepared sUSD3 holder who redeems before sUSD3's accounting is synchronized receives more USD3 than their fair post-loss share. This directly transfers realized losses from the exiting holder to the remaining junior-tranche participants. The protocol's intended loss socialization within the junior tranche is undermined, allowing one sUSD3 holder to exit at the expense of others.
The issue does not create new protocol losses beyond the underlying USD3 loss event, and it does not directly endanger the senior tranche or cause protocol-wide insolvency. The magnitude of the user-to-user transfer depends on the size of the USD3 loss, the sUSD3 TVL, and the timing gap between reports.

Recommendation

Ensure sUSD3 cannot price user actions using stale cached `totalAssets` after USD3 has mutated its USD3 balance. The safest approach is to atomically synchronize sUSD3's accounting whenever `USD3.report()` burns USD3 affecting sUSD3. This can be implemented by having `USD3._postReportHook()` or `_burnSharesFromSusd3()` call a privileged function on sUSD3 that forces an immediate accounting refresh.
Alternatively, modify sUSD3's deposit and redemption flows to detect staleness by comparing `asset.balanceOf(address(this))` with `TokenizedStrategy.totalAssets()` and either refresh the cached value or revert if they diverge. This guards against any external balance mutation, not just those originating from USD3.
Do not rely solely on operational conventions or off-chain keeper sequencing to batch `USD3.report()` and `sUSD3.report()` together, as the contracts do not enforce this invariant and an attacker can backrun a standalone `USD3.report()` transaction before the keeper submits a separate `sUSD3.report()`.

Developer Response

Fixed in 8ab2d6c.
This is addressed by the same stale-accounting patch. `sUSD3.availableWithdrawLimit()` now returns zero before any shutdown/cooldown bypass when either of these conditions is true:

2.6.3 Stale borrower markdowns let USD3 redeemers exit at an inflated NAV

Technical Details

USD3 reports its MorphoCredit position from Morpho market totals, but borrower markdowns are only applied when the defaulted borrower is explicitly touched.
The relevant flow is:

- `src/usd3/USD3.sol:335` calls `morphoCredit.accrueInterest()` during report.
- `src/usd3/USD3.sol:685` values USD3's Morpho position through `expectedSupplyAssets()`.
- `src/libraries/periphery/MorphoBalancesLib.sol:33` updates expected balances for interest, not borrower markdowns.
- `src/MorphoCredit.sol:124` exposes `accruePremiumsForBorrowers()` as the explicit borrower-sync path.
- `src/MorphoCredit.sol:717` updates borrower markdown state only when `_updateBorrowerMarkdown()` is reached.
- `src/MorphoCredit.sol:767` applies markdown deltas to `market.totalSupplyAssets`.

If a borrower is in default but has not been touched, `market.totalSupplyAssets` still carries the debt at face value. USD3 can then run reports against an inflated Morpho NAV. The loss only appears after a later borrower sync and report.

Impact

Medium. Early USD3 redeemers can exit against an inflated NAV after a borrower is economically in default but before the borrower markdown is synchronized. The eventual sync shifts the hidden loss onto remaining USD3 holders and any remaining sUSD3 first-loss backing.

Recommendation

Ensure USD3 cannot report or service withdrawals from Morpho totals that exclude pending borrower markdowns. The team can choose the mechanism: refresh relevant defaulted borrowers, conservatively discount unsynced defaulted debt, or gate reports/redemptions when markdown freshness cannot be established.

Developer Response

We agree that this is an issue, but we don't intend to fix this in the short-term. We will mitigate operationally with frequent accruals/touches of markdown-ed borrowers. We do believe this is closer to a Medium than a High since it's only present during markdowns and can be reduced by operational policies.

2.6.4 sUSD3 exits can leave USD3 temporarily over-deployed relative to live backing

Technical Details

`USD3._subordinationDeployCapWaUSDC()` computes deployment capacity from the `USD3` balance held by the `sUSD3` contract as subordinate backing. That backing can be created by ordinary `BaseHooksUpgradeable.deposit()` calls into `sUSD3`, or temporarily by direct `USD3` transfers to `sUSD3`.
Once the backing is observed, USD3 deposit, `USD3._tend()`, or `USD3.report()` paths may supply additional `waUSDC` to Morpho through `USD3._deployFunds()`. However, `sUSD3.availableWithdrawLimit()` uses a floor from `sUSD3.getSubordinatedDebtFloorInUSDC()` tied to current borrowed debt, not senior credit already deployed or made immediately borrowable.
If Morpho borrow is still low or zero, `sUSD3` holders may redeem through `BaseHooksUpgradeable.redeem()` and remove the backing even though `USD3` has already supplied liquidity based on it.

Impact

Medium. After the `sUSD3` lock and cooldown expire, withdrawals can reduce live subordinate backing while `USD3` liquidity supplied to Morpho remains deployed until the next `tend()` or `report()` rebalance. If `SUSD3_NOMINAL_BACKING_FLOOR` is unset or too low, and an approved borrower draws before that rebalance, debt can be created against liquidity whose backing has already exited. This increases senior `USD3` loss exposure and violates the intended tranche subordination model. A sufficient nominal floor, where `SUSD3_NOMINAL_BACKING_FLOOR` is at least `DEBT_CAP × MIN_SUSD3_BACKING_RATIO / 10_000`, mitigates this path.

Recommendation

In `sUSD3.getSubordinatedDebtFloorInUSDC()`, replace `totalBorrowAssetsWaUSDC` with `max(totalBorrowAssetsWaUSDC, USD3.suppliedWaUSDC())` before converting to `USDC`. This makes the withdrawal floor symmetric with `USD3._subordinationDeployCapWaUSDC()`: the same exposure that enables senior deployment also gates subordinate exits, closing the window between supply and borrow.

Developer Response

Fixed in bfa8c39.

2.6.5 USD3 report can redeploy funds after shutdown

Technical Details

`USD3.report()` reaches `USD3._harvestAndReport()`, which always calls `USD3._tend()` with `asset.balanceOf(address(this))`.

`_tend()` wraps idle USDC into `waUSDC` and supplies local `waUSDC` to Morpho through `USD3._supplyToMorpho()` whenever the effective deployment target exceeds the current deployed amount. There is no `TokenizedStrategy.isShutdown()` guard around this path. The base strategy comments explicitly warn that if reports may be called after shutdown, `_harvestAndReport()` must check the shutdown state so it does not redeploy funds that were freed by emergency withdrawal.

Impact

Medium. Freed assets can be sent back into the credit market during shutdown, making exits dependent on Morpho liquidity again and re-exposing users to credit risk that shutdown was meant to avoid.

Recommendation

In `USD3._harvestAndReport()` and/or `USD3._tend()`, skip wrapping and supplying funds when `TokenizedStrategy.isShutdown()` is true. Post-shutdown reports should only account for assets and optionally withdraw, not redeploy.

Developer Response

Fixed in 9af1400.
The change makes USD3 _tend() a no-op after shutdown, suppresses tendTrigger() during shutdown, and adds regression tests covering post-shutdown report/tend behavior so freed funds are not wrapped or redeployed.

2.7 Low Findings

2.7.1 USD3 can overstate executable withdrawals when waUSDC redemption is capacity-limited

Technical Details

`USD3` services USDC withdrawals by converting the requested amount into a required `waUSDC` quantity and then unwrapping that `waUSDC` back to USDC. The strategy holds `waUSDC` both locally and as a position supplied to Morpho. The withdrawal flow advertises capacity through `availableWithdrawLimit()` and executes withdrawals through `_freeFunds()`.
The implementation mismatch occurs in how these two functions model `waUSDC` redemption capacity:

- `availableWithdrawLimit()` computes redeemable `waUSDC` by summing two separate buckets:
- Local: `min(balanceOfWaUSDC(), WAUSDC.maxRedeem(address(this)))`
- Morpho-backed:
  `min(waUSDCMax, waUSDCLiquidity, WAUSDC.maxRedeem(address(morphoCredit)))`
- `_freeFunds()` performs the actual withdrawal by first pulling `waUSDC` from Morpho to `USD3`, then calling `WAUSDC.redeem(waUSDCToUnwrap, address(this), address(this))` to convert the combined local balance to USDC.

The problem is that `availableWithdrawLimit()` adds two owner-specific redemption caps as if each operates independently, but `_freeFunds()` performs redemption only from `address(this)` after combining local and Morpho-sourced `waUSDC`. If the `WAUSDC` wrapper enforces a redemption capacity limit that binds on the total balance at `address(this)`, the advertised limit can exceed what `USD3` can actually unwrap in one transaction.
For example, if `WAUSDC` has a global redemption cap of 100e6 and `USD3` holds 100e6 locally plus 100e6 supplied to Morpho, `availableWithdrawLimit()` will report that 200e6 USDC is withdrawable. However, when a user attempts to withdraw that full amount, `_freeFunds()` will withdraw 100e6 `waUSDC` from Morpho, bringing the local balance to 200e6, and then attempt to redeem 200e6 `waUSDC` from `address(this)`. If the wrapper's global cap is still 100e6, the `WAUSDC.redeem()` call will revert.
This inconsistency is observable when `WAUSDC.maxRedeem()` enforces a meaningful constraint beyond the trivial paused state, such as a shared liquidity limit or per-address throttle.

Impact

Low. Users and integrators can be told through `maxWithdraw()` that a certain USDC amount is available for withdrawal, but the actual `withdraw()` transaction will revert when `_freeFunds()` attempts to unwrap more `waUSDC` than the wrapper permits at `address(this)`. This causes wasted gas and temporary withdrawal liveness failure. Users can still access their funds by requesting a smaller withdrawal amount that fits within the true redemption capacity, or by waiting for wrapper liquidity to improve. No assets are stolen and no persistent state corruption occurs, but the advertised withdrawal limit does not match the executable withdrawal path.

Recommendation

Modify `availableWithdrawLimit()` to compute a conservative lower bound that reflects what `_freeFunds()` can actually execute. Do not sum two separate owner-based `maxRedeem` values and assume the combined amount can be redeemed by `USD3`. Instead, after accounting for the `waUSDC` that can be withdrawn from Morpho, apply a single aggregate redemption cap at `address(this)`:

```
uint256 localWaUSDC = balanceOfWaUSDC();
uint256 morphoWaUSDC = Math.min(waUSDCMax, waUSDCLiquidity);
uint256 totalRedeemableWaUSDC = Math.min(
    localWaUSDC + morphoWaUSDC,
    WAUSDC.maxRedeem(address(this))
);
```

Additionally, in `_freeFunds()`, cap `waUSDCToUnwrap` by `WAUSDC.maxRedeem(address(this))` after any Morpho withdrawal to gracefully handle scenarios where the wrapper's redemption capacity has changed between view and execution:

```
uint256 waUSDCToUnwrap = Math.min(
    Math.min(localWaUSDC, waUSDCNeeded),
    WAUSDC.maxRedeem(address(this))
);
```

This ensures that the advertised withdrawal limit is always achievable in the actual withdrawal path and prevents reverts when wrapper redemption capacity binds.

Developer Response

Patched in 52bf532.

2.8 Gas Savings Findings

None.

2.9 Informational Findings

None.

3Jane Money Market Contracts - USD3/sUSD3 Update

Completed 2026-05-28