PRELIMINARY REPORT

SHERLOCK

Security Review For 3Jane

Private Best Efforts Audit Contest Prepared For: **3Jane**
Lead Security Expert: **Obsidian**
Date Audited: **October 7 - October 17, 2025**

Introduction

3Jane is a credit-based money market on Ethereum enabling unsecured lines of credit underwritten against verifiable proofs of crypto & bank assets, future cash flows, and credit scores. 3jane issues lines of credit via a morpho blue fork, takes deposits via 4626 vaults, and incentivizes with $JANE.

Scope

Repository: 3jane-protocol/moneymarket-contracts

Audited Commit: 0e570373c79de9fc4a30b891ceb3207d9e14b63f

Final Commit: 4f84d41bf4b20805e2f548532048561052bfdbaa

Files:

- src/CreditLine.sol
- src/Helper.sol
- src/InsuranceFund.sol
- src/interfaces/IProtocolConfig.sol
- src/irm/adaptive-curve-irm/AdaptiveCurveIrm.sol
- src/jane/Jane.sol
- src/jane/PYTLocker.sol
- src/jane/RewardsDistributor.sol
- src/libraries/periphery/MorphoCreditLib.sol
- src/libraries/periphery/MorphoCreditStorageLib.sol
- src/libraries/periphery/MorphoStorageLib.sol
- src/libraries/ProtocolConfigLib.sol
- src/MarkdownController.sol
- src/MorphoCredit.sol
- src/Morpho.sol
- src/ProtocolConfig.sol
- src/usd3/base/BaseStrategyUpgradeable.sol
- src/usd3/interfaces/ISUSD3.sol
- src/usd3/sUSD3.sol
- src/usd3/USD3.sol

Final Commit Hash

4f84d41bf4b20805e2f548532048561052bfdbaa

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

Issues Found

High	Medium
1	7

@@NOTACKNOWLEDGEDTABLE@@

Security experts who found valid issues

0xB4nkz
0xSlowbug
0xaxaxa
0xc0ffEE
0xpetern
0xsh
ChaosSR
ExtraCaterpillar
HeckerTrieuTien
JeRRy0422
Obsidian
PUSH0
Sa1ntRobi
SarveshLimaye
Sparrow_Jac
Varun_05
WillyCode20
X0sauce
Yaneca_b
Ziusz
algiz
axelot
bbl4de
befree3x
dobrevaleri
holtzzx
illoy_sci
ke1caM
lodelux
m3dython
oxwhite
pollersan
roshark
shiazinho
silver_eth
theweb3mechanic
tinnohofficial
tobi0x18
vivekd
wickie
y4y

Issue H-1: Loss of all YT yield accrued due to PYT-Locker staleness

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/35

Found by

PUSH0, axelot, holtzzx, ke1caM, tinnohofficial

Summary

TODO; Also todo: make the report cleaner

Description

In the `PYTLocker`, users can deposit Pendle Yield tokens, with the purpouse of locking them up to expiry.

```
/**
 * @notice Deposits PYT tokens and locks them until expiry
 * @param pytToken The PYT token to deposit
 * @param amount The amount to deposit
 * @dev Tokens must be whitelisted and not expired
 */
function deposit(address pytToken, uint256 amount) external nonReentrant
↪  onlySupported(pytToken) {
    if (amount == 0) revert ZeroAmount();
    if (isExpired(pytToken)) revert TokenExpired();

    // Transfer tokens from user
    IERC20(pytToken).safeTransferFrom(msg.sender, address(this), amount);

    // Update balances
    balances[pytToken][msg.sender] += amount;
    totalSupply[pytToken] += amount;

    emit Deposit(pytToken, msg.sender, amount);
}

/**
 * @notice Withdraws PYT tokens after they have expired
 * @param pytToken The PYT token to withdraw
 * @param amount The amount to withdraw
 * @dev Can only withdraw after token expiry
 */
function withdraw(address pytToken, uint256 amount) external nonReentrant {
```

```
    if (amount == 0) revert ZeroAmount();
    if (!isExpired(pytToken)) revert TokenNotExpired();

    // Update balances (will revert due to underflow if insufficient balance)
    /*
    @audit
    YT are yield bearing tokens (grows over time with yield and fee), balances here
    ↪  are stored by how much the user deposited, and do not increase with the
    ↪  yield accrued,
    This causes loss, of funds, as the user can only withdraw what they deposited,
    ↪  without accountinf for fee.
    */
    balances[pytToken][msg.sender] -= amount;
    totalSupply[pytToken] -= amount;

    // Transfer tokens to userP
    IERC20(pytToken).safeTransfer(msg.sender, amount);

    emit Withdraw(pytToken, msg.sender, amount);
}
```

The problem, whoever, is that Pendle Yield Tokens (YT token) are yield-bearing token tokens, that mantain a global index, tracking how much underlying SY yield should be tracked to an address by the time they got the token to the time they will claim/withdraw.

You can see this in the Pendle YT docs:

https://docs.pendle.finance/pendle-v2/ProtocolMechanics/YieldTokenization/YT

The address that owns the YT tokens during the accrual period will earn the yield.

Meaning users who deposit into the PYTLocker, making the PYTLocker the owner of the YT tokens will earn the accrual period.

If you did not understand, we deposit YT into PYTLocker, and the ownership of YT goes from us to the PYTLocker.

...

The problem, however, is that the locker (owner of YT locked tokens) never actually claims any accrued yield tokens.

When expiration happens (The only time the users can withdraw tokens), the value of the YT token is actually 0 dollars.

This is because, the YT price is determined by future yield (When all yield has been realized, which is maturity, the price is exactly 0).

At maturity/expiration, all yield that was accumulated on the locker can not be claimed.

Impact

At deposit -> withdraw, the PYT locker holds all yield, but at withdraw, the yield hold from YT is completely forgotten and not transfered to the user, in effect, the depositor will lose all the accrued yield.

And, the accrued yield from YT tokens that were on the locker can not be claimed or rescued. They will stay there forever (There isn't a rescue function)

Mitigation

We, at withdraw, need to redeem all pending yield from the locker and transfer it via pro-rata for the user.

This is done by

1: Possibly mantaining a struct of how many YT each user deposited and the total all users deposit 2: Then, having a way to claim underlying yield owned by the PYTLocker at time of withdrawing and transfering the yield to the user

Issue M-1: ISSUE H-4 from Sherlock Audit is not fixed

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/14

Found by

0xB4nkz, 0xc0ffEE, 0xsh, ExtraCaterpillar, Obsidian, X0sauce, bbl4de, dobrevaleri, illoy_sci, ke1caM, lodelux, oxwhite, pollersan, shiazinho, silver_eth, theweb3mechanic, tinnohofficial, tobi0x18

Summary

The lock period extension vulnerability identified in the previous Sherlock audit remains unfixed.

Attackers can maliciously extend any user's lock/commitment period in both USD3 and sUSD3 contracts by depositing minimal amounts (1 wei) through the whitelisted Helper contract, effectively trapping user funds indefinitely at negligible cost.

Root Cause

In both USD3.sol#L::_preDepositHook and sUSD3.sol#L::_preDepositHook, the deposit hooks unconditionally reset lock/commitment periods for any valid deposit. The Helper contract is whitelisted in both contracts, allowing third-party deposits that bypass the intended self-deposit protection.

Internal Pre-conditions

1. Helper contract must be whitelisted as a depositor in both contracts

External Pre-conditions

1. Victim must have existing positions in USD3 and/or sUSD3
2. Victim's lock/commitment period must be nearing expiration

Attack Path

USD3 Attack:

1. attacker calls `helper.deposit(1, victim, false)`
2. Helper calls `USD3.deposit(1, victim)` as whitelisted depositor
3. USD3's `_preDepositHook` extends victim's commitment period to full duration

4. Victim's withdrawal ability is delayed by another commitment period

sUSD3 Attack:

1. attacker calls `helper.deposit(1, victim, true)`
2. Helper flows through USDC → USD3 → sUSD3 deposit path
3. sUSD3's `_preDepositHook` extends victim's lock period to full duration
4. Victim's withdrawal ability is delayed by another lock period

Impact

Users suffer indefinite fund locking through repeated micro-deposit attacks.

PoC

-

Mitigation

Check if the caller of `helper.deposit` is equal to receiver or hardcode it to msg.sender.

Issue M-2: Default, and subsequent jane burn can be front run in case transfers are allowed.

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/19

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xaxaxa, HeckerTrieuTien, Obsidian, PUSH0, WillyCode20, X0sauce, befree3x, lodelux, m3dython, roshark, silver_eth, theweb3mechanic

Summary

In case a borrower defaults his jane token balance will be burned. When a borrower enters default, he is prohibited to transfer his jane tokens. However he can simply transfer tokens from his wallet before entering default to prevent token burn.

Root Cause

The jane token implements a transfer lock in case the borrower is currently in default or failing his repayments:

```
function isFrozen(address borrower) external view returns (bool) {
    if (!markdownEnabled[borrower]) return false;

    // Check if borrower is actually delinquent or in default
    (RepaymentStatus status,) =
    ↪  MorphoCreditLib.getRepaymentStatus(IMorphoCredit(morphoCredit), marketId,
    ↪  borrower);
    return status == RepaymentStatus.Delinquent || status ==
    ↪  RepaymentStatus.Default;
}
```

https://github.com/sherlock-audit/2025-10-3jane/blob/main/moneymarket-contracts/src/MarkdownController.sol#L139

This however does not prevent the borrower to transfer his jane token right before entering this state, bypassing the whole freeze functionality.

Internal Pre-conditions

Jane transfers are activated or a staking contract is approved.

External Pre-conditions

User holds jane tokens. User borrowers. User is about to default.

Attack Path

Before entering `Delinquent` state, stake jane or transfer tokens to different wallet.

Impact

Burn mechanism is bypassed, making the whole markdown system obsolet.

PoC

No response

Mitigation

Allow transfer to only accrue if borrower has no open borrow positions, regardless of repayment status.

Issue M-3: sUSD3 backing math overstates debt and can freeze withdrawals

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/25

Found by

0xc0ffEE, 0xpetern, ExtraCaterpillar, HeckerTrieuTien, Obsidian, SaIntRobi, X0sauce, Ziusz, algiz, axelot, dobrevaleri, illoy_sci, lodelux, shiazinho, silver_eth, theweb3mechanic, tinnohofficial, tobi0x18, wickie

Summary

The sUSD3 contract tries to express market debt in USDC for its backing checks, but after correctly converting waUSDC shares to USDC it mistakenly passes that same USDC value through `USD3.convertToAssets` as if it were USD3 shares. So the share price gets applied again and the required backing becomes too large when the USD3 share price is above 1, which makes `availableWithdrawLimit` return 0 and block withdrawals even though the real backing target is already met.

Root Cause

In sUSD3.sol:424-450, the cap helper converts waUSDC shares to USDC (correct) and then mistakenly runs that USDC through `USD3.convertToAssets`, which expects USD3 shares, so the share price is applied a second time and the number becomes too large:

```
function getSubordinatedDebtCapInUSDC() public view returns (uint256) {
    USD3 usd3 = USD3(address(asset));

    (,, uint256 totalBorrowAssetsWaUSDC,) = usd3.getMarketLiquidity();
    // @audit Wrong applying USD3.convertToAssets to a USDC amount (double
    ↪  application of share price)
    uint256 actualDebtUSDC =
        IStrategy(address(asset)).convertToAssets(usd3.WAUSDC().convertToAssets(tot
        ↪  alBorrowAssetsWaUSDC));

    IProtocolConfig config =
    ↪  IProtocolConfig(IMorphoCredit(morphoCredit).protocolConfig());
    uint256 debtCap = config.config(ProtocolConfigLib.DEBT_CAP);

    uint256 potentialDebtUSDC;
    if (debtCap > 0) {
        // @audit Wrong converting USDC with USD3.convertToAssets as if it were
        ↪  USD3 shares
```

```
        potentialDebtUSDC = IStrategy(address(asset)).convertToAssets(usd3.WAUSDC()
        ↪  .convertToAssets(debtCap));
    }
    uint256 maxDebtUSDC = Math.max(actualDebtUSDC, potentialDebtUSDC);

    if (maxDebtUSDC == 0) {
        return 0;
    }

    uint256 maxSubRatio = maxSubordinationRatio(); // e.g., 1500 (15%)

    // Cap on subordinated debt = max(actual, potential) * subordination ratio
    return (maxDebtUSDC * maxSubRatio) / MAX_BPS;
}
```

In sUSD3.sol:457-473, the floor helper repeats the same pattern: it computes debt in USDC from waUSDC shares (correct) and then feeds it to `USD3.convertToAssets` as if it were USD3 shares, inflating the minimum required backing:

```
function getSubordinatedDebtFloorInUSDC() public view returns (uint256) {
    uint256 backingRatio = minBackingRatio();
    if (backingRatio == 0) return 0;

    USD3 usd3 = USD3(address(asset));

    (,, uint256 totalBorrowAssetsWaUSDC,) = usd3.getMarketLiquidity();
    // @audit Wrong second conversion inflating debt by applying USD3 share price
    uint256 debtUSDC =
        IStrategy(address(asset)).convertToAssets(usd3.WAUSDC().convertToAssets(tot
        ↪  alBorrowAssetsWaUSDC));

    return (debtUSDC * backingRatio) / MAX_BPS;
}
```

Internal Pre-conditions

1. The USD3 share price is greater than 1 because the strategy has accumulated yield since the last reprice.
2. The minimum sUSD3 backing ratio is configured to a non‑zero value or the tranche ratio is set by governance.
3. The credit market has outstanding debt, meaning total borrowed assets in waUSDC shares are non‑zero.

External Pre-conditions

None.

Attack Path

1. Governance calls `ProtocolConfig.setConfig(MIN_SUSD3_BACKING_RATIO, 2000)` to require a 20% minimum backing for sUSD3.
2. Keeper calls the normal harvest flow so USD3 accrues yield and its share price rises above 1 due to earnings being reflected in total assets.
3. Protocol maintains an active USD3 market with outstanding borrow (i.e., `totalBorrowAssetsWaUSDC > 0`) so the backing checks engage.
4. User initiates a withdrawal during a valid window by calling `ITokenizedStrategy.withdraw` (or `redeem`) after starting cooldown.
5. Contract computes the minimum required backing in `availableWithdrawLimit` via `getSubordinatedDebtFloorInUSDC`, converting waUSDC shares to USDC (correct) and then mistakenly converting that USDC again with `USD3.convertToAssets` (incorrect), which inflates the floor.
6. Contract compares current sUSD3 assets against the inflated floor and, finding them not sufficient, returns zero as the withdrawable amount so the user cannot withdraw.

Impact

sUSD3 withdrawals can be blocked when the USD3 share price is above 1, preventing users from withdrawing even though the intended backing requirement is actually satisfied. The risk policy becomes effectively stricter than intended, keeping capital locked in sUSD3 unnecessarily and degrading user liquidity until corrected.

Mitigation

Use only one conversion, by just converting waUSDC shares directly to USDC:

```
function getSubordinatedDebtCapInUSDC() public view returns (uint256) {
    USD3 usd3 = USD3(address(asset));

    (,, uint256 totalBorrowAssetsWaUSDC,) = usd3.getMarketLiquidity();
-    uint256 actualDebtUSDC =
-        IStrategy(address(asset)).convertToAssets(usd3.WAUSDC().convertToAssets(tot
 ↪  alBorrowAssetsWaUSDC));
+    uint256 actualDebtUSDC =
 ↪  usd3.WAUSDC().convertToAssets(totalBorrowAssetsWaUSDC);

    // [...]

    if (debtCap > 0) {
-        potentialDebtUSDC = IStrategy(address(asset)).convertToAssets(usd3.WAUSDC()
 ↪  .convertToAssets(debtCap));
```

```
+       potentialDebtUSDC = usd3.WAUSDC().convertToAssets(debtCap);
    }

    // [...]
}

function getSubordinatedDebtFloorInUSDC() public view returns (uint256) {
    // [...]
-    uint256 debtUSDC =
-        IStrategy(address(asset)).convertToAssets(usd3.WAUSDC().convertToAssets(to
↪  talBorrowAssetsWaUSDC));
+    uint256 debtUSDC = usd3.WAUSDC().convertToAssets(totalBorrowAssetsWaUSDC);

    return (debtUSDC * backingRatio) / MAX_BPS;
}
```

Issue M-4: Stale markdown burn baseline causes accelerated JANE burns on later defaults

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/70

Found by

Obsidian, PUSH0, Sparrow_Jac, Ziusz, algiz, vivekd

Summary

When a borrower exits `Default`, the `MarkdownController` does not reset its burn baseline. On a later default, burns continue from the old, larger snapshot, so JANE burns progress faster relative to the borrower's current holdings.

Root Cause

The `burnJaneProportional` snapshots `initialJaneBalance` on first burn and accumulates `janeBurned`, but only resets on settlement via `burnJaneFull`. There's no reset when the borrower returns to Current. In MarkdownController.sol:153-160:

```
function burnJaneProportional(...) {
    // [...]
    if (initialBalance == 0) {
        initialBalance = jane.balanceOf(borrower);
        if (initialBalance == 0) return 0;
        initialJaneBalance[borrower] = initialBalance; // <-- Snapshot set once and
        ↪  persists across episodes
    }

    // [...]
}
```

And in MarkdownController.sol:196:

```
function burnJaneFull(...) {
    // [...]
    janeBurned[borrower] = 0; // <-- Reset only happens on settlement
    initialJaneBalance[borrower] = 0;
}
```

MorphoCredit detects `Default` entry/exit and calls proportional burn during default, but never resets the controller's baseline when a borrower exits `Default`. In MorphoCredit.sol:738-742:

```
function _updateBorrowerMarkdown(Id id, address borrower) internal {
    // [...]
    bool isInDefault = status == RepaymentStatus.Default && statusStartTime > 0;
    bool wasInDefault = lastMarkdown > 0;

    if (isInDefault && !wasInDefault) {
        emit EventsLib.DefaultStarted(id, borrower, statusStartTime);
    } else if (!isInDefault && wasInDefault) {
        // @audit Doesn't reset controller baseline
        emit EventsLib.DefaultCleared(id, borrower);
    }

    // [...]

    // @audit Burn during default
    IMarkdownController(manager).burnJaneProportional(borrower, timeInDefault);

    // [...]
}
```

Internal Pre-conditions

1. Markdown enabled for the borrower.
2. Borrower defaults at least once so `initialJaneBalance` is set and `janeBurned` accumulates.
3. Borrower exits `Default` (status becomes `Current`) without a settlement (so `burnJaneFull` isn't called).
4. Borrower later defaults again with a materially smaller JANE balance.

External Pre-conditions

None.

Attack Path

1. On the first default, Alice has 10,000 JANE. Controller snapshots `initialJaneBalance` is 10,000 and after some time, `janeBurned` is 1,000.
2. Alice repays obligations, leaves `Default` (status: `Current`) and controller state persists (no reset).
3. Alice sells most JANE and now holds 500 JANE.
4. On the next default, `initialJaneBalance` is still 10,000 and `janeBurned` is 1,000.

5. As time increases, new burns are computed against 10,000. Relative to her 500 JANE, burns are accelerated and can wipe the smaller balance quickly.

Impact

Borrowers who re□default after reducing holdings can lose their remaining JANE much faster than proportional to current holdings. This is a direct, irreversible loss under normal flows.

Mitigation

Add a controller reset:

```
function resetBorrowerState(address borrower) external onlyMorphoCredit {
    janeBurned[borrower] = 0;
    initialJaneBalance[borrower] = 0;
}
```

And call the reset on episode boundaries:

```
function _updateBorrowerMarkdown(Id id, address borrower) internal {
  // [...]
  if (isInDefault && !wasInDefault) {
+     IMarkdownController(manager).resetBorrowerState(borrower);
      emit EventsLib.DefaultStarted(id, borrower, statusStartTime);
  } else if (!isInDefault && wasInDefault) {
+     IMarkdownController(manager).resetBorrowerState(borrower);
      emit EventsLib.DefaultCleared(id, borrower);
  }
}
```

Issue M-5: borrower is retroactively charged premium if his rate switches from 0 to a non-zero value

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/87

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

lodelux, silver_eth

Summary

Because MorphoCredit::premium.lastAccrualTime is only set during the first borrow, users who perform multiple borrows and later have their premium rate increased from 0 to a non-zero value will be charged the updated premium rate retroactively, starting from the timestamp of their first borrow.

Root Cause

Borrowers get charged premium interest when `_accrueBorrowerPremium` is called which does:

```
    function _accrueBorrowerPremium(Id id, address borrower) internal {
        (RepaymentStatus status,) = getRepaymentStatus(id, borrower);


        BorrowerPremium memory premium = borrowerPremium[id][borrower];
@>      if (premium.rate == 0 && status == RepaymentStatus.Current) return;


        if (position[id][borrower].borrowShares == 0) return;


        // Calculate current borrow assets
        Market memory targetMarket = market[id];
        uint256 borrowAssetsCurrent =
        ↪  uint256(position[id][borrower].borrowShares).toAssetsUp(
            targetMarket.totalBorrowAssets, targetMarket.totalBorrowShares
        );


        // Calculate premium and penalty accruals
        uint256 premiumAmount = _calculateOngoingPremiumAndPenalty(id, borrower,
        ↪  premium, status, borrowAssetsCurrent);
```

```
        // Calculate penalty if needed (handles first penalty accrual)
        uint256 penaltyAmount = _calculateInitialPenalty(id, borrower, status,
        ↪  borrowAssetsCurrent, premiumAmount);

        uint256 totalPremium = premiumAmount + penaltyAmount;

        // Skip if below threshold
        if (totalPremium < MIN_PREMIUM_THRESHOLD) {
            return;
        }

        // Apply the premium
        _updatePositionWithPremium(id, borrower, totalPremium);

        // Update timestamp
@>      borrowerPremium[id][borrower].lastAccrualTime = uint128(block.timestamp);
    }

struct BorrowerPremium {
    uint128 lastAccrualTime;
    uint128 rate;
    uint128 borrowAssetsAtLastAccrual;
}
```

and the actual premium amount is calculated using the `borrowerPremium` struct by applying the rate to the elapsed time since `lastAccrualTime`.

Notice however in the above function that this accrual timestamp is only updated at the end of the function and if the borrower has a rate of 0 the function returns early.

everytime this function is called, it is always followed by:

```
    function _snapshotBorrowerPosition(Id id, address borrower) internal {
        BorrowerPremium memory premium = borrowerPremium[id][borrower];

        Market memory targetMarket = market[id];

        uint256 currentBorrowAssets =
        ↪  uint256(position[id][borrower].borrowShares).toAssetsUp(
            targetMarket.totalBorrowAssets, targetMarket.totalBorrowShares
        );
```

```
        // Update timestamp if:
        // - Not initialized (safety check), OR
        // - This is the first actual borrow (transition from 0 debt to positive
        ↪  debt)
@>      if (premium.lastAccrualTime == 0 || (premium.borrowAssetsAtLastAccrual == 0
↪   && currentBorrowAssets > 0)) {
            premium.lastAccrualTime = uint128(block.timestamp);
        }


        // Update borrow amount snapshot
        premium.borrowAssetsAtLastAccrual = currentBorrowAssets.toUint128();


        // Write back to storage
        borrowerPremium[id][borrower] = premium;
    }
```

which also updated the accrual timestamp, but crucially it is not done on subsequent borrows. This means that the user will be retroactively charged a premium if he matches this scenario:

- borrower has a premium rate equal to 0
- borrows x tokens at timestamp T_1 (which are premium-free), `premium.lastAccrualTime == T_1`
- at T_2 he borrows again y tokens (also these should be premium-free), still `premium.lastAccrualTime == T_1`
- at T_3 his premium rate increase to `I != 0`, now the next call to `_accrueBorrowerPremium` goes on to calculate a premium with this new rate but starting from T_1 instead of T_3 borrower is effectively retroactively charged a premium for assets that he borrowed while he had 0 premium rate.

Internal Pre-conditions

1. borrower starts with 0 premium rate
2. he borrows at least twice with this 0 rate
3. borrower gets his rate increased

External Pre-conditions

None

Attack Path

- borrower has a premium rate equal to 0
- borrows x tokens at timestamp T_1 (which are premium-free), `premium.lastAccrualTime == T_1`
- at T_2 he borrows again y tokens (also these should be premium-free), still `premium.lastAccrualTime == T_1`
- at T_3 his premium rate increase to `I != 0`, now `_accrueBorrowerPremium` goes on to calculate a premium with this new rate but starting from T_1 which amounts to `(x + y) * I * (T_3 - T_1)`

Impact

Users will see their loans incur in much higher interest rates than they expected as it is fair for them to assume that any loan taken with a 0% interest rate will not incur in any interest even if their rate will increase later.

PoC

please paste this test and the helper function in `PremiumIntegrationTest.sol` and run it with `yarn test:forge  --match-test testPremiumRateUpgradeAccruesRetroactively -vv`

```
function testPremiumRateUpgradeAccruesRetroactively() public {
    uint256 supplyAmount = 20_000e18;
    uint256 firstBorrow = 5_000e18;
    uint256 secondBorrow = 2_500e18;
    uint256 creditLineAmount = 20_000e18;
    uint128 premiumRatePerSecond = uint128(uint256(0.1e18) / 365 days); // 10% APR
    ↪  = max DRP

    MorphoCredit morphoCredit = MorphoCredit(address(morpho));

    vm.prank(SUPPLIER);
    morpho.supply(marketParams, supplyAmount, 0, SUPPLIER, "");

    _setCreditLineWithPremium(BORROWER, creditLineAmount, 0);

    //  BORROWER takes first loan, with 0% premium rate
    vm.prank(BORROWER);
    morpho.borrow(marketParams, firstBorrow, 0, BORROWER, BORROWER);

    (uint128 initialAccrualTime,, uint128 initialBorrowedAssets) =
    ↪  morphoCredit.borrowerPremium(id, BORROWER);

    _continueMarketCycles(id, block.timestamp + 30 days);
```

```
    // BORROWER takes second loan, again with 0% rate
    vm.prank(BORROWER);
    morpho.borrow(marketParams, secondBorrow, 0, BORROWER, BORROWER);

    (uint128 staleAccrualTimeAfterSecond,, uint128 secondBorrowedAssets) =
        morphoCredit.borrowerPremium(id, BORROWER);

    // the lastAccrualTime is not updated, it still refers to the first borrow
    assertEq(uint256(staleAccrualTimeAfterSecond), uint256(initialAccrualTime));
    assertGt(secondBorrowedAssets, initialBorrowedAssets);

    skip(30 days);

    //borrower sees his premium rate incresaed to a non-zero value, so he should
    ↪  start accruing premium interests from this moment onward.
    _setCreditLineWithPremium(BORROWER, creditLineAmount, premiumRatePerSecond);

    // again, the last acrrual timestamp is still the ts of the first borrow
    (uint128 staleAccrualAfterRateChange,,) = morphoCredit.borrowerPremium(id,
    ↪  BORROWER);
    assertEq(uint256(staleAccrualAfterRateChange), uint256(initialAccrualTime),
    ↪  "timestamp should remain stale");

    // on the same ts of the new rate increase, he gets charged an interest
    ↪  calculated with the new rate but starting from the first borrow!!!
    vm.recordLogs();
    morphoCredit.accrueBorrowerPremium(id, BORROWER);
    uint256 premiumAmount = _retrievePremiumFromLogs();
    // in reality he should be charged no premium at all, as the new rate has just
    ↪  been set!
    uint256 approxRetroactivePremium = (firstBorrow + secondBorrow) *
    ↪  premiumRatePerSecond * (30 days + 30 days) / 1e18;
    assertApproxEqRel(premiumAmount, approxRetroactivePremium, 0.01e18, "premium
    ↪  should be approx. equal to expected");
}

function _retrievePremiumFromLogs() internal returns (uint256 premiumAmount) {
    Vm.Log[] memory entries = vm.getRecordedLogs();

    bytes32 premiumAccruedSig =
    ↪  keccak256("PremiumAccrued(bytes32,address,uint256,uint256)");

    for (uint256 i; i < entries.length; ++i) {
        Vm.Log memory log = entries[i];
        if (log.topics[0] != premiumAccruedSig) continue;
        if (log.topics.length < 3 || address(uint160(uint256(log.topics[2]))) !=
        ↪  BORROWER) continue;

        (premiumAmount,) = abi.decode(log.data, (uint256, uint256));
```

```
    }
}
```

Mitigation

No response

Issue M-6: USD3 is not compliant with EIP-4626 since it does not consider the pause state of WAUSDC

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/92

Found by

0xpetern, ChaosSR, Obsidian, SarveshLimaye, dobrevaleri, tobi0x18, y4y

Summary

According to the contest Readme, `USD3` is supposed to comply with EIP-4626. However, it is not compliant with EIP-4626 since the `maxDeposit`, `maxMint`, `maxWithdraw`, `maxRedeem` functions violate one of MUST statements.

Root Cause

Per EIP-4626:

> **maxDeposit** MUST return the maximum amount of assets deposit would allow to be deposited for receiver and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary). This assumes that the user has infinite assets, i.e. MUST NOT rely on balanceOf of asset. MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0.

> **maxMint** MUST return the maximum amount of shares mint would allow to be deposited to receiver and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary). This assumes that the user has infinite assets, i.e. MUST NOT rely on balanceOf of asset. MUST factor in both global and user-specific limits, like if mints are entirely disabled (even temporarily) it MUST return 0.

> **maxWithdraw** MUST return the maximum amount of assets that could be transferred from owner through withdraw and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary). MUST factor in both global and user-specific limits, like if withdrawals are entirely disabled (even temporarily) it MUST return 0.

> **maxRedeem** MUST return the maximum amount of shares that could be transferred from owner through redeem and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

> MUST factor in both global and user-specific limits, like if redemption is entirely disabled (even temporarily) it MUST return 0.

When users deposit/mint, the `USD3` contract wraps provided `USDC` to `waUSDC`. When users withdraw/redeem, the `USD3` contract unwraps necessary amount of `waUSDC`. The waUSDC contract can be paused and wrapping/unwrapping to/from `waUSDC` is reverted when it is paused.

However, in the availableDepositLimit and availableWithdrawLimit functions which are invoked in the `maxDeposit/maxMint` and `maxWithdraw/maxRedeem` functions, it does not return `0` when `waUSDC` is paused.

Therefore, MUST statements of EIP-4626 are violated and `USD3` is not compliant with EIP-4626.

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

1. `waUSDC` is paused.
2. User calls the `deposit/mint` function but it reverts.
3. User calls the `withdraw/redeem` function but it reverts.

Impact

The `USD3` contract is not compliant with EIP-4626.

PoC

N/A

Mitigation

N/A

Issue M-7: market underestimates totalAvailable liquidity leading to multiple issues

Source: https://github.com/sherlock-audit/2025-10-3jane-judging/issues/131

Summary when a users markdown is updated, the markdown assets are removed from the totalSupplyAssets so the loss starts distribution , this however leads to the market and its integrations underestimating the total available assets even to sub 0

Root Cause total available assets is underestimated across contracts Morpho (withdraw and borrow functions) USD3 (getMarketLiquidity - used in multiple places)

Internal Pre-conditions none

External Pre-conditions none

Attack Path assume totalSupplyAssets = 100 totalBorrowAssets = 95

1. borrowers debt is markdown by 10 new totalSupplyAssets = 100
   actual available collateral = 100 - 95 = 5 virtual available as calculated = 90 - 95 = -5(reverts since this is a uint)
2. borrowers debt is markdown by 3 new totalSupplyAssets = 97
   actual available collateral = 100 - 95 = 5 virtual available as calculated = 97 - 95 = 2 (no revert but there are actually 5 free assets not 2)

Impact free funds cannot be withdrawn break of eip4626 compliance (maxRedeem,maxWithdraw) may revert despite both never being supposed to be able to revert borrowing is unnecesarily limited potential dos of core functionalities in SUSD3 like withdrawals and deposits

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.