SHERLOCK

Security Review For 3Jane

Collaborative Audit Prepared For: **3Jane**
Lead Security Expert(s): **Kirkeelee**
mstpr-brainbot
Date Audited: **August 4 - August 20, 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. This unlocks a three-dimensional collateral space within crypto financial markets by introducing future-backed loans alongside existing asset-backed loans. 3Jane integrates onchain address credit scoring models via Cred Protocol and Blockchain Bureau with offchain VantageScore 3.0 credit scores via zkTLS, enabling risk-adjusted underwriting at scale. To maintain protocol solvency, 3Jane operates onchain auctions where U.S. collections agencies can bid on non-performing debt. By expanding access to unsecured credit, 3Jane unlocks a new era of capital efficiency, empowering a new class of high-productivity economic actors – including cryptonative sole proprietors, businesses, and AI agents – to borrow against future productivity.

Scope

Repository: 3jane-protocol/3jane-morpho-blue

Audited Commit: c92f1cc8a2b45e419fddfeb7cae5d42bb06ed539

Final Commit: 0bea461e7a4a3be3ce969c23ce6c185df6e1a907

Files:

- src/CreditLine.sol
- src/Helper.sol
- src/InsuranceFund.sol
- src/irm/adaptive-curve-irm/AdaptiveCurveIrm.sol
- src/MarkdownManager.sol
- src/MorphoCredit.sol
- src/ProtocolConfig.sol

Repository: 3jane-protocol/usd3

Audited Commit: 4160262e91c7b0af5966d387eb4a2a27c0147e0f

Final Commit: c426a277f3e46ad91a6d1cb3d32a366c00068152

Files:

- src/base/BaseStrategyUpgradeable.sol
- src/sUSD3.sol
- src/USD3.sol

Findings

Each issue has an assigned severity:

- 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.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
7	5	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Settlement flow double deduction and incorrect balance clearing.

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/12

Summary

The `MorphoCredit` settlement flow contains critical accounting vulnerabilities that can lead to double deduction of assets and shares from market balances, as well as incorrect clearing of borrower positions when partial settlements occur. The flow incorrectly assumes full position settlement and fails to coordinate properly between repayment and settlement operations.

Vulnerability Detail

The vulnerability is in the credit line settlement flow through two distinct but related issues:

Issue 1: Double deduction of market balances: When the credit line calls `repay()` followed by `settleAccount()`, the market's `totalBorrowAssets` and `totalBorrowShares` are decremented twice for the same debt amount. The `repay()` function already reduces these totals, but `_applySettlement()` reduces them again without accounting for prior repayments.

Issue 2: Incorrect position clearing for partial settlements: The `_applySettlement()` function always clears the borrower's entire position `(position[id][borrower].borrowShares = 0)` regardless of whether the settlement covers the full remaining balance or only a partial amount. This incorrectly removes borrower shares that should remain active.

Issue 3: Revert on full repayment before settlement: If the repayment in the first step clears the entire borrower balance, `settleAccount()` will revert because `writtenOffShares == 0`, preventing the credit line from completing the settlement process even when this is a valid scenario.

The problematic flow occurs as follows:

Credit line calls `repay()` which decrements `totalBorrowAssets` and `totalBorrowShares` Credit line calls `settleAccount()` which calls `_applySettlement()` `_applySettlement()` decrements the same totals again using the original position values `_applySettlement()` clears the entire borrower position regardless of partial settlement amount

Impact

This vulnerability creates accounting inconsistencies that can lead to protocol insolvency and incorrect user balance tracking, potentially allowing exploitation of the accounting errors for financial gain.

Code Snippet

https://github.com/sherlock-audit/2025-08-3jane/blob/1d8b5f3d10116c04900c8664ea605546c5631732/3jane-morpho-blue/src/MorphoCredit.sol#L812-L868

Tool Used

Manual Review

Recommendation

Redesign the settlement flow to properly coordinate between repayment and settlement operations.

Issue H-2: Accrual timestamp start is always wrong

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/13

Summary

When users borrows, their first accrual timestamp will not be correct because the timestamp is not updated when when snapshotting the borrowers position.

Vulnerability Detail

As seen, when the borrower's credit is first issued, since the timestamp of the user is going to be `0`, it will be set to the `block.timestamp`: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L415-L417

When the user borrows for the first time, since the borrower has no shares, this will early return: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L311

Then, when the user's borrow is finalized, it will be snapshotted via this function, which, since the last accrual is not `0`, won't be updated: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L348-L369

So what happens here is that the user borrowed successfully, but their timestamp didn't change. Their timestamp is still the timestamp from when the credit was first ever issued, even though they didn't borrow until now. This will result in a huge interest accrual the first time interest is going to be accrued.

Coded PoC:

```
// forge test --match-test test_WrongTimestampOnPremium -vv
    function test_WrongTimestampOnPremium() public {
        // Set base rate to 10% APR
        uint256 baseRateAPR = 0.1e18; // 10% in WAD
        configurableIrm.setApr(baseRateAPR);

        // Supply liquidity
        vm.prank(SUPPLIER);
        morpho.supply(marketParams, 10_000e18, 0, SUPPLIER, "");

        // Set up borrower with 10% premium (20% total APR)
        uint256 premiumAPR = 0.1e18; // 10% in WAD
        uint256 premiumRatePerSecond = premiumAPR / 365 days;

        vm.prank(address(creditLine));
        creditLine.setCreditLine(id, BORROWER, 10_000e18,
        ↪  uint128(premiumRatePerSecond));
```

```
        (uint256 lastAccrualTime,,) = morphoCredit.borrowerPremium(id, BORROWER);
        console.log("Premium timestamp:", lastAccrualTime);
        console.log("Block timestamp:", block.timestamp);
        skip(86400 * 7);

        // Borrow 1000 tokens
        uint256 borrowAmount = 1000e18;
        vm.prank(BORROWER);
        morpho.borrow(marketParams, borrowAmount, 0, BORROWER, BORROWER);

        (lastAccrualTime,,) = morphoCredit.borrowerPremium(id, BORROWER);
        console.log("Premium timestamp:", lastAccrualTime);
        console.log("Borrowed block timestamp:", block.timestamp);

        assertNotEq(lastAccrualTime, block.timestamp);
    }
```

Copy this test to PremiumCompoundingTest.sol

Impact

High because of the unfair interest accrual will result on losses for the borrowers

Code Snippet

Tool Used

Manual Review

Recommendation

update the timestamp on _snapshotBorrowerPosition function

Issue H-3: When premium rate is changed for a borrower the timestamp is not correctly updated

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/14

Summary

Timestamp is not updated when the rate is updated, which will lead to double accounting of premium fees on the next accrual.

Vulnerability Detail

When the borrower's rate already exists and they have borrowed shares, the interest will be correctly updated here and the timestamp as well: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L410

However, at the end of the function, the memory `premium` variable will be reassigned to storage, and the accrual timestamp of the memory `premium` is not updated since it's in memory: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L419

This will result in the interest being accrued, but the timestamp is not updated for the user. So, the next time, the same interest will be applied again to the user with the new rate!

Coded PoC:

```
// forge test --match-test test_UpdatePremiumRate_Timestamp -vv
    function test_UpdatePremiumRate_Timestamp() public {
        // Set base rate to 10% APR
        uint256 baseRateAPR = 0.1e18; // 10% in WAD
        configurableIrm.setApr(baseRateAPR);

        // Supply liquidity
        vm.prank(SUPPLIER);
        (, uint256 sharesMinted) = morpho.supply(marketParams, 10_000e18, 0,
        ↪  SUPPLIER, "");
        console.log("Shares minted: ", sharesMinted);

        // Set up borrower with 10% premium (20% total APR)
        uint256 premiumAPR = 0.1e18; // 10% in WAD
        uint256 premiumRatePerSecond = premiumAPR / 365 days;

        vm.prank(address(creditLine));
        creditLine.setCreditLine(id, BORROWER, 10_000e18,
        ↪  uint128(premiumRatePerSecond));
```

```
        // Borrow 1000 tokens
        uint256 borrowAmount = 1000e18;
        vm.prank(BORROWER);
        morpho.borrow(marketParams, borrowAmount, 0, BORROWER, BORROWER);

        skip(86400 * 7);

        (uint256 lastAccrualTime,,uint256 borrowAssetsAtLastAccrual) =
        ↪  morphoCredit.borrowerPremium(id, BORROWER);
        console.log("Premium timestamp:", lastAccrualTime);
        console.log("Borrow assets at last accrual:", borrowAssetsAtLastAccrual);

        // Update premium rate
        vm.prank(address(creditLine));
        creditLine.setCreditLine(id, BORROWER, 10_000e18,
        ↪  uint128(premiumRatePerSecond * 2));

        // Get new premium rate
        (lastAccrualTime,, borrowAssetsAtLastAccrual) =
        ↪  morphoCredit.borrowerPremium(id, BORROWER);
        console.log("Premium timestamp:", lastAccrualTime);
        console.log("Borrow assets at last accrual:", borrowAssetsAtLastAccrual);
    }
```

Copy this test to PremiumCompoundingTest.sol

Impact

Code Snippet

Tool Used

Manual Review

Recommendation

As in the previous issue, move the timestamp set stuff to `_snapShotBorrowerPosition` function

Issue H-4: Lock period extension attack via third-party micro-deposits.

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/19

Summary

External actors can maliciously extend any user's lock/commitment period by depositing minimal amounts (1-2 wei) on their behalf in both USD3 and sUSD3 contracts, effectively trapping user funds indefinitely at negligible cost.

Vulnerability Detail

Both USD3 and sUSD3 contracts implement identical flawed patterns where deposit hooks unconditionally reset lock/commitment periods regardless of who initiates the deposit. In sUSD3, the `_preDepositHook` function resets the lock period to the full 90-day duration (based on test configurations) whenever any deposit occurs for a user. Similarly, USD3 resets the 7-day commitment period (based on test configurations) for any deposit.
https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/usd3/src/sUSD3.sol#L170-L175
https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/usd3/src/USD3.sol#L426-L430

Since `deposit()` and `mint()` functions allow deposits on behalf of any address without validation, attackers can wait until a victim's lock/commitment period is nearly expired and deposit minimal amounts (1-2 wei) to reset the entire balance lock for the full duration again. The minimum deposit check only applies to first-time depositors, making existing users vulnerable to micro-deposit attacks that can be repeated indefinitely.

The attack works across both contracts simultaneously, allowing coordinated fund trapping where victims can be locked in multiple protocols with different time periods using the same low-cost attack pattern.

Impact

Users can have their funds locked indefinitely through repeated micro-deposits across both contracts. The attack cost is approximately 1-2 wei plus gas fees to lock user funds for extended periods, with sUSD3 providing 90-day locks and USD3 providing 7-day locks. Attackers can coordinate these attacks during market stress when liquidity access is most critical, and can front-run withdrawal attempts to extend lock periods just before users try to exit their positions.

Code Snippet

https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/usd3/src/USD3.sol#L411-L430

Tool Used

Manual Review

Recommendation

Implement identical protection in both contracts to prevent third-party period extensions by validating that only the user themselves(and Helper contract) can extend their own lock/commitment periods.

Issue H-5: Griefing by depositing waUSDC to USD3 behalf of sUSD3

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/22

Summary

If sUSD3 has a deposit, then its commitment time will exist, which means that when users withdraw from sUSD3, the transfer hook might fail if the commitment time is not over for sUSD3.

Vulnerability Detail

When users withdraw from sUSD3, the USD3 in the sUSD3 will be transferred to the user. This means the transfer hook will be executed on USD3, which checks whether the commitment time is over for the ”from” account or the depositTimestamp is ”0”. Since sUSD3 just holds the USD3, it never has a depositTimestamp. However, anyone can deposit with ”receiver” being sUSD3 to initiate a depositTimestamp for sUSD3. In that case, when users try to withdraw from the sUSD3, the transfer hook would fail because sUSD3’s commitment time is not over. Anyone can do this permanently by depositing small amounts with receiver sUSD3 to permanently DOS the sUSD3 withdrawals.

https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/usd3/src/USD3.sol#L489-L492

Impact

Permanent DoS for sUSD3 depositors.

Code Snippet

Tool Used

Manual Review

Recommendation

```
- if (to == sUSD3) return;
+ if (to == sUSD3 || from == sUSD3) return;
```

Issue H-6: Fee recipient supplyShares stuck behind usd3-only withdraw.

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/23

Summary

Fee recipient shares are credited as `supplyShares` but cannot be withdrawn because `_beforeWithdraw` only permits calls from the usd3 contract.

Vulnerability Detail

When premiums and base interest are applied, the contract mints `feeShares` into `position[id][feeRecipient].supplyShares`, but `_beforeWithdraw` only allows withdrawals initiated by usd3 (it reverts if msg.sender != usd3). If fees are expected to be withdrawn via normal supply/withdraw flow, those `feeShares` become unwithdrawable and effectively stuck.

https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L586-L592

Impact

Fees credited to the fee recipient can be locked in contract state and unreachable, causing lost fee revenue and incorrect accounting for supply balances.

Code Snippet

https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L586-L592

Tool Used

Manual Review

Recommendation

Permit the `feeRecipient` to redeem credited `supplyShares` by allowing withdraws for the `feeRecipient` in `_beforeWithdraw`.

Issue H-7: MorphoCredit funds can be fully drained

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/25

Summary

Anyone can create markets on the current Morpho fork code and set the credit line to their own custom contracts. With a malicious chain of activities, an attacker could drain all funds supplied to the Morpho fork at no cost.

Vulnerability Detail

First, the attacker will create a market where the `creditLine` contract is malicious, which we will come to. https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/Morpho.sol#L149

Then, the attacker will give his account an infinite amount of collateral for his market id: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L383-L391

Now, the attacker needs to find a way to increase his market's total supply assets so that he can borrow the amount. Since supply is only possible via the `supply()` function, which has the `require` check for `msg.sender == usd3`, it's not possible for the attacker to fund his market. Even if he could have done that, he would still need to supply funds.

However, there is another way where the attacker can increase the `totalAssets` of the market id, which is by creating markdown. First, the attacker will post an obligation for his account and wait some time until his account is in the Default period.

Now that the attacker is in the Default state, the attacker will call the `accrueBorrowerPremium` function. Since the attacker has no borrows, no premium will be issued. However, when it comes to the `_updateBorrowerMarkdown` function, the attacker can create fake markdown with his controlled `markdownManager` contract. Note that this contract is directly fetched from the creditLine contract, which is also attacker-controlled. Let's say the attacker returned a huge amount for markdown. What will happen is that the markdown of the market will be updated, but since the `totalSupplyAssets` is `0`, it will remain `0`, while `totalMarkdown` will be a huge amount.

Then, the attacker will call the `accrueBorrowerPremium` again, but this time he will make the markdown return to `0`. Now, the delta markdown will be `0 - hugeAmount = -hugeAmount`. When the delta is negative, the markdown amount is reset and added to `totalSupplyAssets`! Now, the attacker has managed to increase the `totalSupplyAssets` of his market without adding any assets. Remember, previously the attacker also had an infinite amount of collateral. Basically, now the attacker can borrow the entire supplied funds from other markets!

```
/// @notice Update a borrower's markdown state and market total
/// @param id Market ID
```

```
/// @param borrower Borrower address
function _updateBorrowerMarkdown(Id id, address borrower) internal {
    address manager = ICreditLine(idToMarketParams[id].creditLine).mm();
    if (manager == address(0)) return; // No markdown manager set

    uint256 lastMarkdown = markdownState[id][borrower].lastCalculatedMarkdown;
    (RepaymentStatus status, uint256 statusStartTime) =
        _getRepaymentStatus(id, borrower, repaymentObligation[id][borrower]);

    // Check if in default and emit status change events
    bool isInDefault = status == RepaymentStatus.Default && statusStartTime > 0;
    bool wasInDefault = lastMarkdown > 0;

    if (isInDefault && !wasInDefault) {
        emit EventsLib.DefaultStarted(id, borrower, statusStartTime);
    } else if (!isInDefault && wasInDefault) {
        emit EventsLib.DefaultCleared(id, borrower);
    }

    // Calculate new markdown
    uint256 newMarkdown = 0;
    if (isInDefault) {
        uint256 timeInDefault = block.timestamp > statusStartTime ? block.timestamp
        ↪  - statusStartTime : 0;
        newMarkdown =
            IMarkdownManager(manager).calculateMarkdown(borrower,
            ↪  _getBorrowerAssets(id, borrower), timeInDefault);
    }

    if (newMarkdown != lastMarkdown) {
        // Update borrower state
        markdownState[id][borrower].lastCalculatedMarkdown = uint128(newMarkdown);

        // Update market totals - use a separate function to avoid stack issues
        _updateMarketMarkdown(id, int256(newMarkdown) - int256(lastMarkdown));

        emit EventsLib.BorrowerMarkdownUpdated(id, borrower, lastMarkdown,
        ↪  newMarkdown);
    }
}

/// @notice Update market totals for markdown changes
/// @param id Market ID
/// @param markdownDelta Change in markdown (positive = increase, negative =
↪  decrease)
function _updateMarketMarkdown(Id id, int256 markdownDelta) internal {
    if (markdownDelta == 0) return;

    Market memory m = market[id];
```

```
    // Track total markdowns for reporting/reversibility
    if (markdownDelta > 0) {
        // Markdown increased - add to total
        m.totalMarkdownAmount = (m.totalMarkdownAmount +
        ↪  uint256(markdownDelta)).toUint128();
    } else {
        // Markdown decreased - subtract from total (with underflow protection)
        uint256 decrease = uint256(-markdownDelta);
        m.totalMarkdownAmount =
            m.totalMarkdownAmount > decrease ? (m.totalMarkdownAmount -
            ↪  decrease).toUint128() : 0;
    }

    // Directly adjust supply assets
    if (markdownDelta > 0) {
        // Markdown increased - reduce supply
        m.totalSupplyAssets = m.totalSupplyAssets > uint256(markdownDelta)
            ? (m.totalSupplyAssets - uint256(markdownDelta)).toUint128()
            : 0;
    } else {
        // Markdown decreased (borrower recovering) - restore supply
        m.totalSupplyAssets = (m.totalSupplyAssets +
        ↪  uint256(-markdownDelta)).toUint128();
    }

    market[id] = m;
}
```

Impact

All funds supplied to the morpho fork is in danger.

Code Snippet

Tool Used

Manual Review

Recommendation

Best thing to do here is to make ”createMarket” function permissionned. The risk protocol taking it by making it permissionless is not worth it considering the new functionalities added to morpho fork.

Discussion

fp-crypto

fixed: https://github.com/3jane-protocol/3jane-morpho-blue/pull/44

Issue M-1: When settlement occurs the borrowers collateral and timestamp is not resetted

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/15

Summary

When the borrower is settled borrowers premium timestamp and the collateral is not resetted.

Vulnerability Detail

As seen here, when the settlement occurs, all the variables of the borrower are reset, but the collateral and premium timestamp are not. This means the borrower can re-borrow. Also, the timestamp needs to be cleared as well so that the next time the borrower borrows, the timestamp is correctly updated.
https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L842-L844

Impact

Since the admin can set the credit line ”0” for the borrower the impact is not much for the resetting collateral part. However, time timestamp resetting can't be done by the admin. Overall, should both delete the collateral and the accrual timestamp to minimize admin mistakes.

Code Snippet

Tool Used

Manual Review

Recommendation

Reset collateral amount and the timestamp of the borrower

Issue M-2: Subordination limit on withdrawals in USD3 can be called multiple times

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/17

Summary

There is a subordination invariant between USD3 and sUSD3 however this invariant is not correctly applied.

Vulnerability Detail

Assume: Total USD3 supply = 100 Total USD3 in sUSD3 = 20 Max subordination ratio = 20%

As seen, the sUSD3 is full because the subordination ratio is forced. Also, assume the underlying Morpho liquidity is there. https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/usd3/src/USD3.sol#L332-L388

Now we will trace these values in `availableWithdrawLimit` for USD3:

```
if (sUSD3 != address(0)) {
    uint256 usd3TotalSupply = TokenizedStrategy.totalSupply();

    // sUSD3 holds USD3 tokens, so we check USD3 balance of sUSD3
    uint256 susd3Holdings = TokenizedStrategy.balanceOf(sUSD3);

    // Get max subordination ratio from ProtocolConfig
    uint256 maxSubRatio = maxSubordinationRatio();

    // If maxSubRatio = 1500 (15%), then minUSD3Ratio = 8500 (85%)
    uint256 minUSD3Ratio = MAX_BPS - maxSubRatio;

    // USD3 circulating (not held by sUSD3) must be at least minUSD3Ratio of total
    uint256 usd3Circulating = usd3TotalSupply - susd3Holdings;
    uint256 minUSD3Required = (usd3TotalSupply * minUSD3Ratio) / MAX_BPS;

    // Prevent withdrawals that would drop circulating USD3 below minimum
    if (usd3Circulating <= minUSD3Required) {
        availableLiquidity = 0; // No withdrawals allowed
    } else {
        uint256 maxWithdrawable = usd3Circulating - minUSD3Required;
        availableLiquidity = Math.min(
            availableLiquidity,
            maxWithdrawable
        );
    }
}
```

Say Alice wants to redeem 10 USD3, which is the max amount she can withdraw:

usd3TotalSupply = 100 susd3Holdings = 10 maxSubRatio = 2000 (20%) minUSD3Ratio = 8000 (80%) usd3Circulating = 90 minUSD3Required = 80 availableLiquidity = 90 - 80 = 10

After Alice redeems 10 USD3, the USD3 total supply will be 90. Let's trace again:

usd3TotalSupply = 90 susd3Holdings = 10 maxSubRatio = 2000 (20%) minUSD3Ratio = 8000 (80%) usd3Circulating = 80 minUSD3Required = 72 availableLiquidity = 80 - 72 = 8

So Alice can redeem another 8 USD3. Basically, she can keep doing this in a loop.

In theory, the final amount Alice can withdraw in total is: At each step: `availableWithdrawLimit = 0.2 * S - H` Withdraw `w = 0.2 * S - H` New supply `S' = S - w = 0.8 * S + H` Here, `H = 10`, so the fixed point is `S* = H / (1 - 0.8) = 50`. You converge toward 50.

Which I believe should be the formula inside `availableWithdrawLimit`.

So: usd3TotalSupply = 50 susd3Holdings = 10 maxSubRatio = 2000 (20%) minUSD3Ratio = 8000 (80%) usd3Circulating = 40 minUSD3Required = 40 availableLiquidity = 40 - 40 = 0

Impact

Not sure this is intended behaviour. If it is, its not an issue but if it is then its broken functionality.

Code Snippet

Tool Used

Manual Review

Recommendation

Use the formula stated in vuln detail section. If its intended no fix needed.

Issue M-3: Loss amount is not correctly calculated in _postReportHook

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/18

Summary

When there is a loss on USD3 report the losses will be removed from sUSD3 if it can absorb the loss. However, the amount to burn from sUSD3 is not correctly calculated.

Vulnerability Detail

Assume at `t = 0`: `totalSupply = 100` `totalAssets = 100`

At `t = 10`, `report` is called with a realized loss. Since the hook is called **after** the report, `totalAssets` and `totalSupply` are already updated in storage.

Say the loss is `20`. After the report: `totalSupply = 100` `totalAssets = 80`

Now, tracing inside the `if` statement (conditions met since there's a loss): `susd3Balance = 20` (assume) `sharesToBurn = 20 * 100 / 80 = 12.5`

Then, `12.5` shares will be burned and deducted from the sUSD3 balance, so at the end: `totalSupply = 100 - 12.5 = 87.5` `totalAssets = 80`

As we can see, the loss is not properly corrected. This happens because the report already changes `totalSupply` and `totalAssets`, so `_postReportHook` using the **after** values will not calculate the actual loss correctly.

Impact

Code Snippet

Tool Used

Manual Review

Recommendation

Get the previous values before the report and check what was the total assets and total supply before report and calculate the `sharesToBurn` with the previous values.

Discussion

tapired

Also, if the locked profit is enough to cover the losses then the ”extra” loss will be accounted from sUSD3.

Issue M-4: Min deposit can be bypassed by passing type(uint256).max on USD3

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/21

Summary

Users must need to mint/deposit at least minDeposit amount to USD3. However, this can be bypassed by passing max uint256.

Vulnerability Detail

As seen in TokenizedStrategy.deposit function here, if the amount is passed as type(uint256).max then the balanceOf(msg.sender) is used: https://github.com/yearn/tokenized-strategy/blame/9ef68041bd034353d39941e487499d111c3d3901/src/TokenizedStrategy.sol#L495-L497

which is not checked against the minDeposit. A user can have 1 wei of balance and pass type(uint256).max, bypassing the minDeposit successfully and depositing 1 wei of shares.

https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/usd3/src/USD3.sol#L422-L424

Impact

Broken functionality.

Code Snippet

Tool Used

Manual Review

Recommendation

Add an extra logic when the amount is type(uint256).max. Simply, if it is, check the balanceOf(msg.sender) and compare it with minDeposit

Issue M-5: Borrowers might repay or borrow before the post obligations will lead to inconsistent state

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/24

Summary

If borrower repays some amount before the post obligations transaction the `endingBalance` will be huge and the next penalty the borrower sees can be huge.

Vulnerability Detail

When the obligations are posted, the `endingBalance` is never validated: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L517-L528

As a result, if the borrower repays just before the obligations are posted, several issues can occur:

- The `amountDue` can be higher than the borrower's entire balance, or it could be higher than the normal repayment amount at the end of the cycle.
- If the borrower borrows more before the obligations are posted, the `amountDue` would be lower than it should be.

Additionally, if the borrower repays before the obligations are posted, the penalty in the next cycle could be excessively large if they ever become delinquent. If the borrower borrows more, then the penalty to be paid can become `0` due to how the penalty is calculated: https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L259-L261

Impact

If the borrower borrows more before the obligations are posted, the impact is medium. However, since borrowers are trusted parties, I believe it is unlikely that a borrower would do this intentionally. Still, there could be an unfortunate transaction ordering where the borrower borrows more just before the obligations are posted without acting maliciously. Alternatively, if the obligations-posting transaction remains pending for too long, the borrower could repay or borrow more in the meantime, causing the issue described above.

Code Snippet

Tool Used

Manual Review

Recommendation

Some validation checks would make sense especially in the protocols negative case where user borrows before post obligations. 1- check amountDue is not higher than the borrowers entire debt 2- check endingBalance is higher than the users current borrows assets

Issue L-1: Repayment by shares blocked when obligation exists due to zero assets check.

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/11

Summary

The `MorphoCredit` contract contains a user experience flaw that prevents users from repaying their debt using shares when they have outstanding payment obligations, forcing them to use only asset-based repayment.

Vulnerability Detail

The vulnerability occurs when users attempt to repay using the shares parameter instead of assets directly on the `MorphoCredit` contract not the `Helper` contract. The `_beforeRepay` hook calls `_trackObligationPayment(id, onBehalf, assets)` where assets equals zero when repaying by shares. The obligation tracking function then checks if the zero payment amount is sufficient to cover the obligation, causing a revert even though the shares-based repayment would convert to sufficient assets to cover the debt. The core issue is that obligation payment validation occurs before the shares-to-assets conversion takes place in the main `repay` function, using the raw assets parameter which is zero for shares-based repayments.

Impact

Users with outstanding payment obligations cannot utilize the shares-based repayment method.

Code Snippet

https://github.com/sherlock-audit/2025-08-3jane/blob/1d8b5f3d10116c04900c8664ea605546c5631732/3jane-morpho-blue/src/MorphoCredit.sol#L640-L651

Tool Used

Manual Review

Recommendation

Call the `_beforeRepay` after assets/shares are calculated inside the `repay` function of Morpho.sol.

Issue L-2: Possible underflow on settlement flow

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/16

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

Summary

Vulnerability Detail

Since the `writtenOffAssets` is calculated via rounding UP this operation can underflow if the borrower is the only borrower of the protocol and the settlement is applied.
https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L848

https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/3jane-morpho-blue/src/MorphoCredit.sol#L829

Impact

Code Snippet

Tool Used

Manual Review

Recommendation

check whether the underflow exists if not it exists its because the round up and the borrower is the only borrower. Which means you can safely reset it to ”0” since the underflow happens because of 1wei round up.

Issue L-3: sUSD3 users can withdraw before the sUSD3 report

Source: https://github.com/sherlock-audit/2025-08-3jane/issues/20

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

Summary

When there are losses on USD3 the USD3 balance of sUSD3 will be burnt to compensate the losses for USD3. However, there is a small time window untill the sUSD3 report is called where the sUSD3 losses are not realized which can be used for users to bypass the losses and leave the losses to other sUSD3 depositors.

Vulnerability Detail

https://github.com/sherlock-audit/2025-08-3jane/blob/99c70e1e81fef06fc14cef2ee59317df44796268/usd3/src/USD3.sol#L511-L538 Losses on USD3 are compensated by the sUSD3 as seen here by burning its USD3 balance. However, this loss is not realized for sUSD3 until sUSD3 calls report() as well. If any user has a withdrawal cooldown already passed, they can withdraw without incurring the loss and pass the losses to other sUSD3 depositors.

Impact

Since the cooldown is there the impact is not that serious because its not instantaneous for sUSD3 depositors to withdraw. However, if any user already started their cooldown and it ended and cooldown window period is not over they can swiftly withdraw to dodge losses.

Code Snippet

Tool Used

Manual Review

Recommendation

I think you can also call sUSD3.report() in `_postReportHook` of USD3. This would eliminate any scenario where sUSD3 depositors withdraw without getting the losses.

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.