PRIVATE

Code Assessment

of the Monolith Smart Contracts

April 14, 2026

Produced for

Inverse Finance

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Inverse Finance Team,

Thank you for trusting us to help Inverse Finance with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Monolith according to Scope to support you in forming an opinion on their security risks.

Inverse Finance implements a factory for creating single-collateral stablecoins.

The most critical subjects covered in our audit are precision of arithmetic operations, integration with other protocols and functional correctness. Earlier versions of the protocol had a number of issues across these categories, most of which have been addressed:

On precision of arithmetic, earlier versions used a pool-based redemption design that introduced serious rounding issues (see User Shares deviate from Total Shares) and performed arithmetic at token-native precision, causing economically meaningful rounding errors for uncommon collateral types (see Collateral with High or Low Decimals). Both have been resolved by switching to per-borrower redemptions, adding decimal constraints, and specifying value ranges on supported assets.

On integrations with other protocols, the Monolith integrates with ERC-4626 vaults whose share price, if inflated, could be used to drain protocol reserves (see Inflated PSM vaults). This has been corrected.

On functional correctness, earlier versions contained an incorrect handling of collateral price decimals (see Price is in Wrong decimals) and a reentrancy vulnerability via ERC-777 tokens (see protocol draining with token reentrancy). Both have been resolved. However, debt share prices can be artificially inflated through write-offs, enabling an attack on new depositors (see Debt Share Price Inflation via writeOff). This has been partially addressed and the residual risk accepted.

The general subjects covered are documentation and trustworthiness. Documentation is improvable, since not all edge-case scenarios are formally documented, see Security considerations for Bots. It should be noted that while the factory places no restrictions on which collateral types it can be deployed with, that does not imply that any type of collateral or asset is supported. Deployers of new stablecoin systems must thoroughly investigate the compliance of each deployed stablecoin and its configuration with supported token types.

Security regarding trustworthiness is good, with only minimal trust requirements for operators. The open protocol design puts significant responsibility on users to verify system state and dependencies before entering, as described in Security Considerations for Users. Further, certain parts of the protocol rely on active participation of users, as without it the interest rates can grow unboundedly, see Unbounded Interest Rate Growth. Further, any protocol integrating Monolith's Vault must account for the possibility of share price inflation, see Vault can be inflated.

In summary, we find that the codebase provides a improvable level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	5
• Code Corrected	3
• Specification Changed	1
• Code Partially Corrected	1
Medium-Severity Findings	5
• Code Corrected	5
Low-Severity Findings	13
• Code Corrected	8
• Code Partially Corrected	1
• Risk Accepted	4

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Monolith repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	06 October 2025	38ce8ee77041cf5e06e3dd53dbba5bea1884098d	Initial Version
2	03 November 2025	baf7a7ab1f80e2a95f587933118be3f3f8904f47	Version with fixes
3	23 March 2026	fc049e1fefc3c900cba36acf3da55c8a4c5f850d	Version with fixes
4	14 April 2026	f5a6fa6aabc50543311c58d5c9dc9e5fd98aec47	Final Version

For the solidity smart contracts, the compiler version `0.8.24` was chosen.

The following files are in scope:

```
src/Factory.sol
src/Coin.sol
src/InterestModel.sol
src/Vault.sol
src/Lender.sol
```

2.1.1 Excluded from scope

Any contracts that are not explicitly listed in the scope section are excluded from the scope of this review. All other files in the repository, including other smart contract files and tests, are explicitly excluded from the scope of this review.

Further, while integrations with external components such as Chainlink Oracles, Collateral and PSM Assets / Vaults external assets are in scope of this review, the actual code of these external components is out of scope.

The Monolith in scope integrates with external ERC-4626 vaults. These vaults must be fully compliant with the standard. Any non-compliance must be carefully considered for its impact on Monolith before deployment.

In this report, we assume that the contracts are deployed to Ethereum Mainnet or other EVM-equivalent chains. The correctness of the codebase if deployed on other chains is not in scope of this review.

2.2 System Overview

This system overview describes the last version (Version 4) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Inverse Finance offers a factory for creating single-collateral stablecoins. Each instance consists of a stablecoin (called `Coin`), a core `Lender` contract, and a yield-bearing `Vault` contract.

2.2.1 Factory

The `Factory` contract serves as the entry point for creating new, isolated stablecoin systems. Deploying a new system is completely permissionless and in each deployment a new `Coin` token contract, a `Lender` core contract, and a `Vault` contract are deployed. All deployed `Lender` instances share a single, immutable `InterestModel` contract to calculate interest rates. The factory maintains a registry of all deployed `Lender` contracts. It also includes an operator role responsible for managing protocol-level fees.

2.2.2 Coin

The `Coin` contract is a standard ERC20 token with 18 decimals, representing the stablecoin of a given deployment. New `Coin` tokens are minted by the associated `Lender` contract, when users borrow assets, fees and interest are accrued or `Coin` are bought via the Peg Stability Module (PSM). Any user can burn `Coin` tokens from their own balance. The `Lender` burns assets when debt is repaid or `Coin` are sold via the PSM.

2.2.3 Lender

The `Lender` contract is the central hub for user interactions within each stablecoin system. It manages collateral, debt, liquidations, redemptions, and interest accrual.

2.2.3.1 Dual Debt System

As in similar systems, users provide collateral in form of a single token (`collateral`) and mint stablecoins against it. A key feature of the protocol is its dual-debt system. The default status for borrowers is **Non-Redeemable**. In this status they pay a variable interest rate on their debt but are protected from redemption. Users can borrow stablecoin as long as their debt does not exceed their borrowing power that is calculated with a collateral factor that cannot exceed 85% set during deployment:

$$borrowingPower = price \times collateral \times collateralFactor$$

Further, user's debt must exceed the `minDebt` or be zero. `minDebt` is also a parameter chosen by the creator, subject to a system-wide minimum value.

Alternatively, borrowers can opt into **Redeemable** status. In this case, they pay 0% interest on their debt, but in exchange, their deposited collateral is made available for redemptions by any `Coin` holder.

2.2.3.2 Redemptions

In a redemption, a `Coin` holder exchanges their tokens for the underlying collateral at face value. Only the collateral of "Free Debt" borrowers is subject to redemption. When a user redeems `Coin`, the system repays the debt of a specified free debt borrower and gives the redeemer an equivalent value of the pooled collateral, minus a small `redeemFeeBps` (max 3%). Redemptions that would reduce the borrower's debt below `minDebt` are not allowed.

2.2.3.3 Oracle

The protocol uses a Chainlink oracle to price collateral. The system operates in one of three modes depending on the status of the price feed.

Regular Mode In regular mode, all protocol functions, including borrowing, liquidations, and redemptions, are fully enabled.

Reduce-Only Mode (Stale Price) If the last price update from the oracle is more than 25 hours old, the system enters a "reduce-only" mode to protect against stale data. In this state, no new debt can be issued, and users cannot withdraw collateral when they have outstanding debt. To encourage a safe wind-down, the reported price begins to decay linearly to zero over a 24-hour period. Liquidations and redemptions remain enabled and use this decaying price. Similarly, if the total debt in the system exceeds the collateral value then the system enters reduce-only mode.

No-Liquidation Mode (Failed Oracle) In the event of a critical oracle failure, where the feed reverts or returns a price of zero, the system enters its most defensive state. To prevent failures both liquidations and redemptions are disabled entirely until the oracle recovers.

The protocol can freely switch between these modes as the oracle status changes and maintains no internal state about the current mode it is in.

2.2.3.4 Liquidations and Write-Offs

The protocol includes mechanisms to manage undercollateralized positions. A position becomes liquidatable when its debt exceeds its borrowing power. A liquidator can repay a portion of the borrower's debt in exchange for a corresponding amount of collateral plus a liquidation incentive of up to 10%. The amount to be liquidated is the greater of 25% of the total debt or 10,000 `Coin`. In extreme cases where a position is deeply underwater, a write-off occurs. For this the debt value must be more than 100 times greater than the remaining collateral value. The borrower's entire remaining debt is then socialized pro rata across all other borrowers in the system (both paid and free), and the liquidator receives all the borrower's remaining collateral as a reward.

2.2.3.5 Peg Stability Module (PSM)

Each `Lender` can be optionally deployed with a PSM, which allows users to swap the `Coin` for a specified stable asset (`psmAsset`) at a 1:1 ratio. Users can sell `Coin` for `psmAsset` or buy `Coin` with `psmAsset`. Selling `Coin` is always without fees. Buying `Coin` is free for the first half of the period leading up to the `immutabilityDeadline` that is up to 4 years after deployment, in the second half, a fee ramps up linearly from 0% to 1%. Further, the deployer can set an ERC-4626 compliant vault (`psmVault`) for the `psmAsset`. In that case the `psmAsset` is deposited in the vault to earn interest, and any profits generated can be withdrawn by the local operator.

2.2.4 InterestModel

The variable interest rate is calculated in an external `InterestModel` contract. The contract itself is stateless and solely uses input data from the `Lender` in its calculation. The rate model is designed to maintain a target balance between free and paid debt in the system. For this the **Free Debt Ratio** is calculated as:

$$ratio = \frac{totalFreeDebt + freePsmAssets}{totalPaidDebt + totalFreeDebt + freePsmAssets}$$

The protocol then defines a target range for the free debt ratio [`targetFreeDebtRatioStartBps`, `targetFreeDebtRatioEndBps`]. If the current debt ratio is below the target range, indicating that there is not enough free debt, the interest rate for paid debt increases exponentially to incentivize users to switch to free debt:

$$r(t) = r_0 \cdot e^{\lambda t}$$

If the ratio exceeds the target range, signaling too much free debt, the interest rate decreases exponentially to encourage borrowing in the paid debt pool, following the formula:

$r(t) = r_0 \cdot e^{-\lambda t}$

The parameter λ is defined as $\frac{\ln(2)}{halfLife}$, where `halfLife` is an adjustable parameter between 24 hours and 30 days, controlling the speed of rate adjustments.

2.2.5 Vault

The Vault is an ERC4626-compliant tokenized vault where users can stake their `Coin` to earn yield, thus creating demand for `Coin`. The yield earned is derived from the amount of interest paid by "Paid Debt" borrowers, but the yield can be at most equal to the interest rate paid by borrowers. Any excess interest is instead accrued to the protocol's operator.

2.3 Trust Model

Roles:

- **Deployers:** *Untrusted.* They can deploy a new stablecoin system but cannot alter its configuration. Any user interacting with a new stablecoin system (incl. operators) should verify that the system is configured correctly and has no malicious contracts configured before interacting with it or risk losing their funds.
- **Global Operator (Factory):** *Partially trusted.* This role can set a protocol-wide fee (up to 10%) on interest revenue and designate a fee recipient. The fee must be configured to be greater than 1 BPS or it falls back to default fee of `feeBps`. They cannot upgrade contracts or directly access user funds.
- **Local Operator (Lender):** *Partially trusted before the immutabilityDeadline.* This role can adjust key risk parameters for a specific `Lender` instance, such as the interest rate `halfLife`, target debt ratios, and redemption fees. They can also set a local reserve fee (up to 10%) and can trigger the immutability deadline early. After the deadline, their power is significantly reduced, since they can only change the local fee (up to 10%) and the fee recipient.
- **Manager (Lender):** *Partially trusted.* A less privileged role than the Local Operator, capable of adjusting some risk parameters but not fees. They can also trigger a system shutdown.
- **Delegates:** *Fully trusted by the user delegating to them.* A user can delegate control of their position to another address, which can then withdraw collateral, borrow more `Coin`, or change the debt status on the user's behalf.
- **End Users:** *Untrusted.* They interact with the protocol's public functions under the rules enforced.

External Components:

- **Chainlink Oracles:** *Fully trusted.* The protocol relies on Chainlink to provide accurate and timely price data for collateral. While there are safeguards for stale or failed oracles, the system's solvency depends on the oracle's integrity during normal operation.
- **Collateral:** *Fully trusted.* The protocol assumes that the tokens used as collateral are well-behaved ERC20 tokens without malicious features. Rebasing tokens or fee on transfer are not supported. Reentrant tokens (i.e. ERC-777) are not supported.
- **Optional: PSM Asset:** *Fully trusted.* The protocol assumes that the tokens used by the PSM are not malicious. Further, any exploit of the PSM asset can spill over to the `Coin` as long as the PSM is enabled. A configured PSM Asset cannot be reentrant or have transfer fees. However, in theory it can be rebasing although a loss is not fully handled (see PSM behavior in case of depeg or Insolvency)

- **Optional: PSM Vault:** *Fully trusted.* Any configured PSM Vault must not be malicious. Similar to a PSM asset, any exploit of the PSM asset can spill over to the `Coin`. A configured PSM Vault must not be reentrant or have deposit or withdrawal fees. The Vault has to be fully compliant with ERC-4626 to work. For example `convertToShares` has to round down.

The following **value ranges** are assumed for a correctly configured deployment:

- **Coin:** peg target between $0.001 and $1,000 USD.
- **Collateral:** 1 wei worth more than 10^{-33} USD.
- **PSM Vault Asset:** 1 wei worth less than $0.001 USD.
- **Minimum Debt:** `minDebt` worth at least 1,000 wei of collateral.

2.4 Changes from Version 1:

In Version 1:

- lower bound for `halfLife` was 12 hours instead of 24 hours.
- upper bound for `collateralFactor` was 100% instead of 85%.
- decimals of the collateral token were not constrained.
- minimum debt floor was not present.

2.5 Changes from Version 2:

In Version 2:

- redemptions were pool-based and pro-rata across all free debt borrowers. In Version 3, redemptions target a single borrower directly.
- collateral balances were internally normalized to 18 decimals. In Version 3, they are stored in the collateral token's native decimals.
- liquidators could attempt to repay more than the collateral they receive. In Version 3, a `getMaxRepayByCollateral` cap is enforced.
- the system did not enter reduce-only mode when total debt exceeded total collateral value.
- reentrant tokens were supported.

2.6 Changes from Version 3:

- no value ranges were defined.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Debt Share Price Inflation via writeOff Code Partially Corrected Risk Accepted

Medium-Severity Findings	0
Low-Severity Findings	5

- Redemptions Can Be Front-Run Risk Accepted
- Adjust Can Be Frontrun Risk Accepted
- Function setRedemptionStatus() Creates Phantom Debt Risk Accepted
- System State After All Positions Are Closed Code Partially Corrected Risk Accepted
- Unbounded Interest Rate Risk Accepted

5.1 Debt Share Price Inflation via writeOff

Design High Version 1 Code Partially Corrected Risk Accepted

CS-INVMONO-001

Debt share prices can be artificially inflated, since it is possible to write off large amounts of debt across only a few wei of debt. This can be exploited to attack new depositors, since `increaseDebt` rounds up the number of shares minted, giving up to one extra share to a new depositor. Furthermore, an attacker could exploit the inflated shares by continuously liquidating small amounts of assets from users with inflated debt shares. As `decreaseDebt` rounds down the number of shares burned, the attacker can pay back debt without burning the corresponding shares. The attacker's profit comes from their proportional share of the debt pool, since each liquidation pays back collective debt using the collateral of victim users, while the attacker retains their inflated debt shares.

Attackers can also combine the `writeoff` with an earlier redemption that scales down the debt shares, since scaling will already inflate the debt shares by a factor of 1e18 (from 1e-9 to 1e+9 debt per debt share).

Code partially corrected / Risk accepted:

In Version 2, `increaseDebt` reverts if the excess amount of debt created by the new shares is bigger than a percentage of the amount of coin repaid, which is set by `maxBorrowDeltaBps`.

However, a user can still open a position in a market with inflated debt shares as long as they are not rounded down too much when opening their position. They could then get their collateral stolen with a liquidation.

The Monolith has provided the following reasoning for keeping the code unchanged:

> We plan on notifying users in the UI when there's bad debt in the system, but we only prevent new borrowing if debt exceeds collateral value, as we don't have a good way to detect bad debt existing for individual positions.

5.2 Redemptions Can Be Front-Run

Design | Low | Version 3 | Risk Accepted

CS-INVMONO-036

A redeemable borrower can monitor the mempool and front-run an incoming `redeem` call by switching to paid status via `setRedemptionStatus`. Because the status change is immediate and has no timelock, the redemption reverts. This allows a borrower to enjoy the 0 % interest rate of free debt while avoiding actual redemption of their collateral.

Risk accepted:

Inverse Finance has accepted the risk, but has decided to keep the code unchanged.

5.3 Adjust Can Be Frontrun

Design | Low | Version 1 | Risk Accepted

CS-INVMONO-011

The `adjust` function enforces that a user's debt balance is either 0 or above the `minDebt` threshold.

```
if(debtDelta != 0) require(debtBalance == 0 || debtBalance >= minDebt, "Debt below minimum and larger than 0");
```

Furthermore, `adjust` allows anyone to call it on behalf of another user, provided the call "improves" the user's position by reducing their debt or increasing their collateral.

```
if(collateralDelta >= 0 && debtDelta <= 0) return;
```

This combination creates a griefing vector. For instance, if a user submits a transaction to decrease their debt to exactly `minDebt`, an attacker can front-run it by repaying 1 wei of their debt. This causes the user's debt to fall just below `minDebt`, leading the call to `adjust` to revert.

More critically, a user with redeemable free debt could have a small portion of their debt redeemed by another party and as a result a call to top up their collateral or adjust their debt reverts.

This poses a higher risk for positions managed by other smart contracts such as multisigs or governance systems. Their actions are more predictable and reverts have a higher impact when coupled with other logic.

Risk accepted:

Inverse Finance has accepted the risk, but has decided to keep the code unchanged.

5.4 Function `setRedemptionStatus()` Creates Phantom Debt

Design | Low | Version 1 | Risk Accepted

CS-INVMONO-038

The function `Lender.setRedemptionStatus()` computes a user's debt using `getDebtOf()` which is rounded *up*; then it wipes his debt before changing its redemption status and finally calling `increaseDebt()` with the previously-calculated debt amount. Since `increaseDebt()` also rounds *up*, this might end up increasing the total debt. This operation can be looped many times, in order to amplify its effect. However, no solvency check is performed on the user, who could therefore make himself liquidatable, or even cause bad debt to the system. Furthermore, this additional debt, albeit tracked by the appropriate storage variable, is not detected and treated in the same way that `accrueInterest()` does: it is not represented by freshly-minted coins, assigned to reserves and the Vault

---

Risk accepted:

Inverse Finance has accepted the risk, but has decided to keep the code unchanged.

5.5 System State After All Positions Are Closed

Design | Low | Version 1 | Code Partially Corrected | Risk Accepted

CS-INVMONO-015

Suppose the last position in the system is written off. Since there is no other position to socialize the debt into, it will be completely erased. The `Lender` will then be in a state analogous to its initial state, where the total debt is zero. However, any outstanding coin supply will still exist and can be used if any user joins the system again, meaning the system is insolvent.

Consider the following attack scenario:

1. A malicious user creates a coin.
2. They then deposit a few wei of collateral and take paid debt.
3. Since at that point, all the debt is paid debt, the interest rate starts going up exponentially.
4. The malicious user waits for the debt to grow to 100 times the collateral value. We assume no one goes to liquidate since this is not profitable.
5. The user then takes a bigger position to mint coins and liquidates themselves in such a way that they recover their collateral.
6. Now all debt is written off, but they still hold coins.
7. They then market the coin and wait for other users to invest.

Code partially corrected / Risk Accepted:

The example attack scenario is no longer possible since a minimum debt floor has been introduced in Version 2. However, this can be bypassed using redemptions, or other techniques.

Inverse Finance is aware of the issue and intends to make sure user interfaces warn about dangerous coins.

5.6 Unbounded Interest Rate

Design | Low | Version 1 | Risk Accepted

CS-INVMONO-025

There is no upper limit on the interest rate charged. If manipulated upwards, a malicious actor could liquidate every user of the protocol within a single block. Note that with the specified half-life of 12 hours, the interest rate can not manipulated upwards quickly, however it is still considered to be a best practice to set a maximum interest rate.

Furthermore, note that the function `Lender.accrueInterest` casts the interest rate to an uint88. Although reaching the theoretical upper bound of approximately 309,485,000% APR is economically unrealistic, the cast itself could result in an unexpected truncation.

Version 2:

No upper limit for the interest rate has been implemented. However, a technical limit of approximately 309,485,000% APR to the interest rate has been introduced to avoid overflowing state variables.

Under this interest rate, no safety bounds can be assumed for the debt growth, so debt shares can be inflated with interest rate growth.

Starting at the initial interest rate of 2% annual, assuming the most aggressive half-life of 24 hours, a debt amount can be multiplied by 6.7 trillions after 6 months, and more than 1e18 after 7 months.

$r(t) = r_0 2^{\frac{t}{86400}}$

$d(t) = d_0 \Pi_{t'=0}^{t} (1 + r(t'))^{\frac{1}{365 \cdot 86400}}$

$r_0 = 2\%$

$d(31 \cdot 86400) \approx 1.85 d_0$

$d(93 \cdot 86400) \approx 1411 d_0$

$d(182 \cdot 86400) \approx 6.7 \times 10^{12} d_0$

$d(216 \cdot 86400) \approx 1.78 \times 10^{18} d_0$

Note that this assumes continuous compounding. Under real conditions, the interest rate is only updated every time a user interacts with the system. If no interactions take place, it remains at 2% and the debt is only $d'(182 \cdot 86400) \approx 1.01 d_0$.

Risk accepted:

Inverse Finance responded:

```
We consider coins with runaway interest and no interactions over 6 months to essentially be dead coins.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	4

- Price Is in Wrong Decimals Code Corrected
- PSM Assets Decimals Mismatch Code Corrected
- Protocol Draining With Token Reentrancy Specification Changed
- User Shares Deviate From Total Shares Code Corrected

Medium-Severity Findings	5

- Sell Does Not Account for Vault Fees Code Corrected
- Collateral With High or Low Decimals Code Corrected Specification Changed
- High Collateral Factor Undermine Liquidation Code Corrected
- Inflated PSM Vaults Code Corrected
- Zero Value Debt Shares Code Corrected

Low-Severity Findings	8

- Sell Returns Wrong Value Code Corrected
- Inconsistent Decimals Code Corrected
- Free Debt Can Be Manipulated via PSM Code Corrected
- Redemption Fees Allow Arbitrage Code Corrected
- Rescaling Forgives Debt Code Corrected
- Staleness Threshold Too Strict Code Corrected
- Try-catch Can Be Triggered by OOG Code Corrected
- Vault Is Not Compliant With ERC-4626 Code Corrected

Informational Findings	6

- Unused Variable Code Corrected
- Incorrect Natspec Comments Code Corrected
- Mismatch of Documentation and Code Code Corrected
- Missing Events for Admin Operations Code Corrected
- Old Compiler Version Code Corrected
- Solmate Library Version and Deprecation Code Corrected

6.1 Price Is in Wrong Decimals

Design High Version 2 Code Corrected

CS-INVMONO-027

In Version 2, the `_cachedCollateralBalances` were normalized to 18 decimals to avoid issues with high-decimal collateral tokens. However, the price fetched from `Lender.getFeedPrice()` is still normalized to `36 - collateral.decimals()` decimals.

This creates a decimal mismatch in the borrowing power calculation. The borrowing power, which should be in 18 decimals (like the debt token), will instead be represented in `36 - collateral.decimals()` decimals:

$$borrowingPower = \frac{price \times collateral \times collateralFactor}{1e18 \times 10000}$$

Where:

- `price` is in `36 - collateral.decimals()` decimals
- `collateral` is in 18 decimals
- Result is in `36 - collateral.decimals()` decimals instead of 18

As such, for high-decimal collateral tokens (e.g., 36 decimals), the borrowing power will be severely understated, while for low-decimal collateral tokens (e.g., 6 decimals), the borrowing power will be significantly overstated.

Code corrected:

In Version 3, the normalization to 18 decimals was removed, so collateral balances are once again stored in their native token-decimal precision. Hence, the `borrowingPower` is correctly in 18 decimals again without additional changes.

6.2 PSM Assets Decimals Mismatch

Design High Version 1 Code Corrected

CS-INVMONO-002

The `Lender` assumes that the PSM asset and the psmVault have 18 decimals precision. However, as can be seen in `buy` and `sell` the PSM asset can have arbitrary decimals:

1. The free debt ratio adds the `freePsmAssets`, which are denominated in the PSM asset, to the `totalFreeDebt`, which is denominated in `Coin`. As such the debt ratio is too high when the PSM asset has more than 18 decimals and too low debt ratio when the PSM asset has fewer decimals. That biases the interest rate calculations.
2. More critically, the function `accruePsmProfit` calculates the accrued profit for a PSM Vault by comparing `previewWithdraw` to `freePsmAssets` and later adds the difference to `accruedLocalReserves`, which is then paid out in `Coin` with 18 decimals. Similarly, if no vault is set, it calculates the profit by comparing `balanceOf` with `freePsmAssets`. If the PSM asset or vault has fewer decimals, then the profits of the protocol get underestimated, and funds get stuck. If the PSM asset or vault has more decimals, then the profits of the protocol get overestimated, and excessive withdrawals cause insolvency.

Code corrected:

In Version 2, a new function `normalizePsmAssets` is used to convert PSM asset amounts from their native decimals to the standard 18-decimal precision.

The function `getFreeDebtRatio` normalizes the `freePsmAssets` before adding it to `totalFreeDebt`. And `accruePsmProfit` normalizes PSM asset amounts before adding them to `accruedLocalReserves`.

6.3 Protocol Draining With Token Reentrancy

Security | High | Version 1 | Specification Changed

CS-INVMONO-003

If the chosen collateral token implements the ERC-777 standard, hooks can be automatically called on external contracts when the protocol performs a transfer. This can cause reentrancy vulnerabilities.

In `writeOff()`, the protocol first performs the transfer, and then sets the user's balance to 0. This could allow a malicious user to perform a call to `adjust` during the token callback, thus withdrawing their collateral twice.

Version 2:

The specification was updated to explicitly support reentrant tokens according to EIP-777. Furthermore, the reentrancy vulnerability in `writeOff` was explicitly addressed by switching the order of operations in `writeOff()`, so that the user's balance is set to 0 before the transfer is performed.

However, additional reentrancy vulnerabilities remain in the codebase:

1. Function `liquidate` sends tokens to the *msg.sender* before setting the user's collateral balance to the lower value. A user could liquidate themselves from a smart contract and after getting control over the execution context reenter `adjust` to withdraw their collateral.
2. Function `adjust` pulls tokens from `msg.sender` when `collateralDelta` is positive. A hook is called on sender before moving the tokens, as specified by EIP-777, but after `nonRedeemableCollateral` was increased. Since the redeemable collateral is computed as follows, during the hook the system is in a state where it underestimates the amount of redeemable collateral.

```
collateralToInternal(collateral.balanceOf(address(this))) - nonRedeemableCollateral
```

3. Function buy gives execution context to the `msg.sender` when transferring PSM assets. If the PSM asset is a reentrant token, it would call into the user before any money is deposited into the `psmVault`, causing the profit of the `Lender` to be underestimated. Similarly, `sell` updates the `freePsmAssets` after sending out the tokens, so the profit is underestimated during the callback. Any Monolith reading stale state from the protocol during these callbacks could be at risk (see read-only-reentrancy for reference).

Specification changed:

The Monolith does not supports reentrant tokens in Version 3.

6.4 User Shares Deviate From Total Shares

Design | High | Version 1 | Code Corrected

CS-INVMONO-004

Users can redeem arbitrary amounts of debt from the `Lender` contract. Redeeming debt leads to rescaling of debt shares.

```
if( totalFreeDebtShares / totalFreeDebt > 1e9) {
    epoch++;
    totalFreeDebtShares = totalFreeDebtShares.mulDivUp(1e18,1e36);
    emit NewEpoch(epoch);
}
```

Here `totalFreeDebtShares` gets rounded up when rescaling.

Similar logic is implemented when any borrower is updated:

```
// Loop through missed epochs (max 5 iterations considering max uint256 is 2^256 - 1 would go to zero in 5 iterations)
for (uint i = 0; i < 5 && _borrowerEpoch < epoch && borrowerDebtShares > 0; ++i) {
    ...

    _borrowerEpoch += 1;
    borrowerDebtShares = borrowerDebtShares.divWadUp(1e36) == 1 ? 0 : borrowerDebtShares.divWadUp(1e36); // If shares is 1 round down to 0
}
```

The user's shares are rounded up, but to allow any borrower to exit after multiple redemptions without debt, the borrower is forgiven one share if they have only one share left.

There are two possible results of this rounding:

First scenario: The total number of shares is rounded up, while individual borrowers are rounded down. This leads to a situation where the total shares are higher than the sum of all borrowers' shares. The relative rounding error can become arbitrarily large between the total shares and the sum of user shares, e.g., the total shares in the system are 10, but the sum of all users is only 0. This makes the system insolvent as there is debt remaining after everyone has exited the system. Furthermore, the protocol becomes permanently blocked since everyone could exit and repay all debt without burning the total shares, hence `Lender.increaseDebt` could revert on underflow with `totalFreeDebt` equal to zero.

Second scenario: The total number of shares is rounded down, while individual borrowers are rounded up. This leads to a situation where the total shares are lower than the sum of all borrowers' shares, e.g., the total shares in the system are 10, but the sum of all users is 20. If $1 million gets redistributed, then the users would receive debt worth $2 million.

The underlying issue is that the relative rounding error between total shares and the sum of user shares can become arbitrarily large, since redeemed amount and total shares are arbitrary.

Here is an example showcasing the issue:

`Initial state`:

- Total debt: 9e9
- Total shares: 9e18
- Share price: 1e-9

Now, suppose 8e9 of the debt is redeemed:

`Updated state`:

- Total debt: 1e9
- Total shares: totalFreeDebtShares.mulDivUp(1e18,1e36) = 9

- Share price: 1e9

Afterward the same process is repeated for each user in `updateBorrower()`. Let's assume there are 9 user with each having 1e18 shares. Their shares are now all getting rounded down to 0 shares, since

```
shares = divWadUp(1e18, 1e36) = 1
```

As a result, a total of 1e9 debt is effectively forgiven. This amount can be economically meaningful for high-value debt tokens such as BTC (1e-9 BTC ≈ 0.1 cent).

In combination with inflated debt shares, this issue becomes particularly severe (see Debt Share Price Inflation via writeOff and Unbounded Interest Rate):

1. Create a deviation between total shares and user shares via repeated redemptions as described above. Result: 10 total debt shares exist, but the sum of all user shares is 0.
2. This already inflates the debt shares to 1e9 debt per share.
3. Further inflate the debt share value to 3000 * 1e18 by either writing off debt or accruing interest.
4. Attacker deposits 1 ETH into the system. Another user also enters with 1 ETH and has the redeemable flag set.
5. Attackers borrows 3000 USD and immediately executes a redemption. When the attacker repays 3000 USD and receives 1 ETH of collateral, the protocol reduces the collective debt but doesn't reduce any individual user's collateral balance since all users have 0 debt shares (so `redeemedCollateral` is 0).

The total balance changes is

- Attacker's position: +1 ETH, 1 ETH still unencumbered in the protocol
- Protocol: -1 ETH

The attacker effectively received 1 ETH for free. The system becomes insolvent because it pays out more collateral than it should without reducing the attacker's ETH balance proportionally. This can be repeated to drain the protocol.

Code corrected:

In Version 3, `redeem` was changed to target a specific borrower, reducing that borrower's debt directly via `decreaseDebt`. This removes the need for rescaling `totalFreeDebtShares` and eliminates the rounding divergence between total and per-user shares that enabled the described vulnerabilities.

6.5 Sell Does Not Account for Vault Fees

Design | Medium | Version 2 | Code Corrected

CS-INVMONO-028

The function `Lender.sell()` computes the shares of the user to redeem using `ERC4626.previewDeposit()` as this rounds down the number of shares. Later, it uses these shares to redeem.

However, using the deposit method to get a rounded down estimate of the withdrawable shares assumes deposit and withdrawals are symmetric. If the Vault has deposit or withdrawal fees, the `freePsmAssets` accounting will break and either assume too many or too few assets are redeemed.

Code corrected:

In Version 2, the function was updated to use `convertToShares` to compute the number of shares to redeem. The assets are redeemed directly to the seller, so any withdrawal fees are borne by them.

6.6 Collateral With High or Low Decimals

Design | Medium | Version 1 | Code Corrected | Specification Changed

CS-INVMONO-005

The protocol does not use constant precision for its arithmetic operations. Instead, operations involving collateral have varying precision depending on the collateral token's decimal configuration, leading to potential issues with both low-decimal and high-decimal tokens.

1. Low-Decimal, High-Value Tokens (e.g., WBTC with 8 decimals or GUSD with 2 decimals)

The amount of redeemed collateral (`redeemedCollateral`) is rounded up by one unit in `Lender.updateBorrower()`. For high-value and low-decimal collateral tokens such as WBTC and GUSD, a single wei of loss accounts for an economic loss of up to $0.01 each time a borrower is synced. Similar rounding occurs whenever a user leaves the system, since `Lender.getDebtOf()` rounds up 1 wei. In general, collateral tokens with low decimals worsen the rounding errors in the system. While these errors favor the protocol, they hurt users and can lead to stuck funds.

2. High-Decimal Collateral Tokens with Low Value (e.g., tokens with 36 decimals)

Less critically, the `Lender.getFeedPrice()` function normalizes the collateral price to an internal representation with `36 - collateral.decimals()` decimals. If a collateral token has 36 decimals, the target precision is 0, so for any collateral with a market price below $1.00, the normalized price will be rounded down to `0` and then set to `1` wei by `getCollateralPrice()`, causing incorrect price calculations throughout the system.

Additionally, during redemptions, the redeemed collateral amount is multiplied by `1e36` before dividing by the number of shares. This intermediate multiplication can overflow if more than approximately 100,000 units (with 36 decimals) are getting redeemed.

Version 2:

The code has been updated to normalize all token amounts to a fixed 18-decimal precision before performing internal calculations, thus preventing economically significant rounding errors and overflow risks with standard tokens.

Code corrected / Specification changed:

The 18-decimal normalization introduced in Version 2 has been removed in Version 3. However, the specific sources of the above issues have been addressed by other changes:

1. The pool-based redemption mechanism (including `updateBorrower()` and the rounding of `redeemedCollateral`) was replaced by per-borrower redemptions, eliminating the original source of the 1-wei rounding loss described in issue 1.
2. A `MAX_DECIMALS = 30` constraint was added to the `Lender` constructor, preventing collateral tokens with excessively high decimals. This eliminates the zero-precision and overflow scenarios described in issue 2.

Additionally, strict value bounds on the expected value of collateral tokens were added to the specification (see System Overview), which should prevent significant rounding losses for a correctly configured system.

6.7 High Collateral Factor Undermine Liquidation

Design Medium Version 1 Code Corrected

CS-INVMONO-007

Stablecoin deployers can set the collateral factor (CF) to any value between 0% and 100%. This parameter creates two distinct problems when set too high.

1. Reduced Liquidation Incentives:

To incentivize liquidators, the system offers a bonus of up to 10% on the liquidated collateral. However, this bonus is capped by the total collateral available in the position. If the CF is above 90.9%, a user can borrow an amount of debt so high that the required 10% bonus cannot be fully paid out from their collateral (i.e., `debt * 1.1 > collateral`).

This disincentivizes liquidators, who may choose to only partially liquidate the debt to seize all the available collateral, leaving the position with remaining bad debt.

2. Liquidation attacks:

High collateral factors can create liquidations that worsen the health of the user instead of improving it. An attacker could then use a flash loan to perform a series of self-liquidations and write-off that make every position in the system liquidatable and then proceed to liquidate everyone:

1. **Create Trigger Position:** The attacker creates a small bad debt position that is unattractive for normal liquidators to write-off. This can be done by creating a small debt position and then partially liquidating it, leaving a small amount of unbacked debt and no collateral.
2. **Create Large, Risky Position:** Using a flash loan, the attacker creates a very large position that is on the verge of being liquidatable.
3. **Trigger Self-Liquidation:** The attacker writes off their own "trigger" position. This makes the large, risky position eligible for liquidation.
4. **Liquidate themselves:** The attacker repeatedly liquidates their own large position. Since only 20% of the debt can be liquidated in each round, they need to perform this in multiple iterations, earning the liquidation incentive on each iteration.
5. **Write-off bad debt:** After extracting value via liquidation incentives, the attacker writes off the remaining (now undercollateralized) debt of their large position. This loss is socialized across all other debtors in the system.
6. **Cascade Liquidations:** The socialized loss can push other healthy positions into a liquidatable state. The attacker can then proceed to liquidate these new positions, continuing the cycle.

Code corrected:

The maximum collateral factor has been capped to 85% in Version 2.

6.8 Inflated PSM Vaults

Design Medium Version 1 Code Corrected

CS-INVMONO-008

The protocol allows specifying a PSM Vault during deployment that holds the PSM assets and generates yield. Whenever a user buys `Coin`, the `Lender` deposits the acquired `psmAsset` into the PSM Vault via `Vault.deposit()` and mints shares to the lender. Similarly, when a user sells `Coin`, the `Lender` withdraws the required amount of `psmAsset` from the PSM Vault via `Vault.withdraw()` and burns shares from the lender.

Vault implementations typically round down the number of shares minted and round up the shares burned, so each deposit and withdrawal incurs a tiny loss. This loss can be amplified when the share price in the vault is inflated via a donation attack. An attacker could first inflate the vault's price per share by making a large direct deposit to the vault, then repeatedly trigger deposits and withdrawals through PSM operations to drain the Lender's funds.

Version 2:

The `sell` function has been updated to transfer the rounding loss to the user instead of the Vault.

```
uint256 sharesOut = psmVault.previewDeposit(assetOut);
assetOut = psmVault.redeem(sharesOut, msg.sender, address(this));
freePsmAssets -= assetOut;
require(assetOut >= minAssetOut, "redeem failed");
```

Users can protect themselves by setting an appropriate `minAssetOut` value when calling `sell`. Additionally, some protections have been added to the `buy` function.

```
require(psmVault.totalSupply() > minTotalSupply, "PSM vault total supply below minimum");
uint256 shares = psmVault.deposit(assetIn, address(this));
require(shares > 0, "PSM deposit failed");
```

However, the protection remains insufficient. While reverting when zero shares are minted protects the vault against getting rounded down to zero, the vault can still be rounded down significantly (e.g., from 1.99 to 1 share in the worst case), so the depositor could lose up to 50% of their deposit.

Version 3:

The `sell()` function charges the rounding loss to the user, but `buy()` does not, since `coinOut` amount to mint is computed from assetIn directly, without first correcting for rounding loss.

Code corrected:

In **Version 4**, the `buy()` function was updated to compute the amount of shares to mint based on the actual amount of assets deposited to the vault, instead of the user input. This way, the rounding loss is also charged to the user in case of a deposit, and not just in case of a withdrawal.

```
uint256 redeemableFor = previewRedeemOrConvertToAssets(shares);
freePsmAssets += redeemableFor;
(coinOut, coinFee) = getBuyAmountOut(redeemableFor);
```

6.9 Zero Value Debt Shares

Design | Medium | Version 1 | Code Corrected

CS-INVMONO-006

The `increaseDebt` function checks if `totalFreeDebtShares` (respectively `totalPaidDebtShares`) is zero before performing division. However, this does not prevent division by zero, as the actual denominator in the calculation is `totalFreeDebt` (respectively `totalPaidDebt`). If the total debt becomes zero while the total shares remain non-zero, any subsequent call to `increaseDebt` or `decreaseDebt` will revert.

An attacker can manipulate the system's state to reach this scenario. For redeemable debt, the attacker creates a debt position and redeems all but 1 wei of their debt. When another redeemable user fully repays their position, the rounding behavior in `Lender.decreaseDebt()` can cause `totalFreeDebt` to become zero while the `totalFreeDebtShares` are non-zero.

Similarly, for non-redeemable debt, the attacker creates a position with 1 wei of debt and switches it to non-redeemable status via `Lender.setRedemptionStatus()`. When another non-redeemable user closes their position, `totalPaidDebt` can become zero while the `totalPaidDebtShares` are non-zero.

Once triggered, this prevents new users from borrowing, as `increaseDebt` will revert. Existing users cannot partially repay their debt, as `decreaseDebt` will also revert. However, users can still fully close their positions, since in that code path no division by `totalPaidDebt` or `totalFreeDebt` occurs.

Version 2:

The function now correctly checks and divides by `totalFreeDebt` (respectively `totalPaidDebt`) when calculating the number of shares to mint or burn. This ensures that division by zero cannot occur.

However, the `totalFreeDebtShares` are not reset when `totalFreeDebt` becomes zero, creating "ghost shares" in the system. This means that if a user closes their position, they would not be required to repay all of their debt, but only:

$$debtToRepay = \frac{userDebtShares \times totalFreeDebt}{totalFreeDebtShares}$$

Since `totalFreeDebtShares` includes ghost shares from previous operations, the denominator can be artificially inflated, allowing users to repay less debt than borrowed. The number of ghost shares can be large compared to the user shares when debt shares were previously heavily deflated.

Code corrected:

Debt rescaling has been eliminated in Version 3. As noted in Invariant: debt share prices are always at least 1, the price can now no longer drop below 1.

6.10 Sell Returns Wrong Value

Correctness | Low | Version 3 | Code Corrected

CS-INVMONO-037

In case a PSM vault is configured, the function `sell()` should return the `actualAssetOutToSeller` amount (which is net of vault fees), rather than the `assetAmount` used for intermediate computations.

Code corrected:

The function was corrected in Version 4 to return `actualAssetOutToSeller` from the function.

6.11 Inconsistent Decimals

Design | Low | Version 2 | Code Corrected

CS-INVMONO-029

The `adjust()` function treats its `collateralDelta` argument as having the same number of decimals as the collateral if it is positive, and 18 decimals if it is negative. While this should not impede functionality it increases the risk of user errors and is not documented.

Code corrected:

In Version 3, collateral amounts are no longer normalized to 18 decimals, and the `collateralDelta` in `adjust()` is always treated as having the same decimals as the collateral token.

6.12 Free Debt Can Be Manipulated via PSM

Design | Low | Version 1 | Code Corrected

CS-INVMONO-012

Users can buy `Coin` for `psmAsset` by calling function `Lender.buy()`. Similarly, they can sell `Coin` via `Lender.sell()`. Buying `Coin` increases the `freePsmAssets` by the amount of `psmAsset` sold into the Lender and selling decreases it by the amount of `psmAsset` bought.

The `freePsmAssets` contribute to the free debt ratio, which is again used in `Lender.accrueInterest()`.

```
function getFreeDebtRatio() public view returns (uint) {
    uint _adjustedTotalFreeDebt = totalFreeDebt + freePsmAssets;
    return _adjustedTotalFreeDebt == 0 ? 0 : _adjustedTotalFreeDebt * 10000 / (totalPaidDebt + _adjustedTotalFreeDebt);
}
```

The interest rate model uses a type of PID controller that lowers the interest rate when the amount of free debt goes up and increases the interest rate when the amount of free debt goes down. Therefore, if a user buys stablecoin with PSM assets, the free debt goes up and as a result the interest rate decreases. On the other hand, if the user sells stablecoin, then the free debt goes down, and the interest rate goes up.

However, buying and selling do not call `accrueInterest()` and thereby omit to snapshot the previous interest rate. This allows an attacker to buy and sell large amounts of assets, and then call `accrueInterest()` to manipulate the interest rate in the desired direction. Since selling stablecoin for PSM assets incurs no fee, it can be cheaper to manipulate the interest rate upwards.

Code corrected:

Calls to `accrueInterest()` were added at the beginning of `sell()` and `buy()`.

6.13 Redemption Fees Allow Arbitrage

Design | Low | Version 1 | Code Corrected

CS-INVMONO-009

`Lender.redeem` allows anyone to redeem `collateral` for `Coin` at the current Chainlink price, minus a redemption fee. Under normal circumstances, redemptions are expected to ensure a lower bound for the pegged stablecoin. However, the parameterization of the protocol allows for profitable redemption arbitrage even when the coin is perfectly pegged.

The amount of collateral received for a given amount of Coin after fees is:

$$col = debt \times (1 - f)/\hat{p}$$

Where:

- $\hat{p}$ is the (potentially stale) price from Chainlink's price feed
- p^* is the real market price that the redeemer could achieve by selling the collateral on a DEX
- f is the redemption fee

The profit of the redeemer in units of debt and ignoring gas cost are:

$$profit = col \times p^* - debt = debt \times (1 - f)/\hat{p} \times p^* - debt$$

, which is positive if $\hat{p}/p^* > 1/(1 - f)$.

Under realistic parameters, this condition is frequently met. The maximum redemption fee that can be set is 3%, and Chainlink price feeds of lower market cap coins as envisioned as collateral token for Monolith have higher deviation thresholds than that (i.e. Stargate Token), so redemptions are profitable even when the stablecoin is perfectly pegged:

$$1.05 > \frac{1}{1 - 0.03} = 1.031$$

That makes it expensive to open up redeemable positions and with second order effects on the interest rate and peg.

Code corrected:

In Version 3, Inverse Finance increased the maximum redemption fee to 5%.

6.14 Rescaling Forgives Debt

Design | Low | Version 1 | Code Corrected

CS-INVMONO-013

As soon as `totalFreeDebtShares > 1e9 × totalFreeDebt`, the debt shares are scaled down by a factor of 1e18 to avoid overflows in the debt shares accounting:

```
if( totalFreeDebtShares / totalFreeDebt > 1e9) {
    epoch++;
    totalFreeDebtShares = totalFreeDebtShares.mulDivUp(1e18,1e36);
    emit NewEpoch(epoch);
}
```

This scaling is not an issue for the total debt shares. However, similar logic is implemented for individual borrowers when they are updated:

```
// Loop through missed epochs (max 5 iterations considering max uint256 is 2^256 - 1 would go to zero in 5 iterations)
for (uint i = 0; i < 5 && _borrowerEpoch < epoch && borrowerDebtShares > 0; ++i) {
    ...

    _borrowerEpoch += 1;
    borrowerDebtShares = borrowerDebtShares.divWadUp(1e36) == 1 ? 0 : borrowerDebtShares.divWadUp(1e36); // If shares is 1 round down to 0
}
```

If `1e9 + 1` debt shares are currently worth one unit of debt, then when rescaling happens with division of 1e18, this destroys up to `1e18` debt shares, or up to `(1e18 - 1) / (1e9 + 1)` ≈ `1e9` units of debt that become unbacked by user shares. With Bitcoin at $100,000, this represents approximately $100,000 / 1e9 = $0.0001 or 0.01 cents that gets forgiven per user. However, for higher-value assets, the loss can be more significant.

Code corrected:

As of Version 3, debt rescaling has been removed, therefore this issue is no longer possible.

6.15 Staleness Threshold Too Strict

Correctness | Low | Version 1 | Code Corrected

CS-INVMONO-014

The function `getCollateralPrice()` assumes that price older than `STALENESS_THRESHOLD`, which is set to 25 hours, are stale. However, Chainlink price feeds with a heartbeat exceeding 25 hours exist. Consider for example the AMPL/USD feed, which has a heartbeat of 48 hours. This could cause coins to unwind almost completely at a point where it is not necessary.

Code corrected:

In Version 2, `stalenessThreshold` can be set by the coin creator.

6.16 Try-catch Can Be Triggered by OOG

Security | Low | Version 1 | Code Corrected

CS-INVMONO-016

The Lender contract uses try-catch statements to prevent unexpected reverts from blocking operations. The downside of this approach is that it incentivizes adversaries to make these calls fail by triggering the catch branch that omits logic. The catch statement in Inverse Finance`s Monolith could get triggered in three distinct scenarios by the external call:

1. The call reverts
2. The call panics (i.e. division by zero)
3. The call runs *out of gas* (OOG).

The cases 1. and 2. are hard to exploit, since the Monolith is either calling itself or semi-trusted contracts (i.e. Chainlink price feeds or Collateral Tokens). However, case 3. can be exploited by malicious users, who provide insufficient gas to external calls in order to bypass certain logic. According to EIP-150, a maximum of 63/64 of the remaining gas can be forwarded to a subcall, which means at least 1/64 of the

gas is retained by the caller; some contracts retain even more. An attacker can therefore craft calls that still pass enough gas for the outer call to succeed while leaving the subcall fail with OOG. As such, this is particularly easy if no expensive logic is executed after the try-catch statement.

In `Lender.liquidate()`, the function call `writeOff()` as its final step in a try-catch and then returns. As no expensive logic is executed, it's vulnerable to aforementioned attack. Further `writeOff()` call invokes `getFeedPrice()`, which in turn calls the `decimals()` function on the token contract. Some tokens - such as stETH (Lido: stETH Token) do not forward all available gas to subcalls. Instead, in the case of stETH, 10,000 gas is reserved, allowing the outer call to succeed.

As such an attacker can skip writing off a position after a liquidation.

Code corrected:

In Version 2, the Lender contract records the gas left before a try-catch call, and if the call fails, checks that it was more than a hardcoded threshold.

6.17 Vault Is Not Compliant With ERC-4626

Design | Low | Version 1 | Code Corrected

CS-INVMONO-017

The Vault does not comply with the ERC-4626 standard, since viewer functions `totalAssets`, `convertToShares`, `convertToAssets`, `previewDeposit`, `previewMint`, `previewWithdraw`, `previewRedeem` do not include pending yield. For example the standard specifies for `totalAssets`:

```
SHOULD include any compounding that occurs from yield.
```

Further, `previewDeposit` and `previewWithdraw` do not take into account the `MIN_SHARES` that will be taken from the first depositor. As a result, the result of the deposit() / mint() call will be inconsistent with the preview.

In Version 2, the Vault overrides `totalAssets()` to compute the pending yield. It returns the current balance plus the pending yield.

In Version 3, additional viewer functions `maxDeposit`, `maxMint`, `maxWithdraw`, and `maxRedeem` were added. They are not compliant with the standard:

- `maxDeposit` and `maxMint` do not return `type(uint256).max` even though there is no limit on the assets that may be deposited. Further `maxDeposit()` returns 0 and `maxMint()` returns `type(uint256).max - supply` when the total supply is zero, violating the standard.
- `maxRedeem` and `maxWithdraw` cap the withdrawable assets by the current balance in the vault, without including pending interest payments. However, any call to `withdraw` or `redeem` would first realize interest, making more assets available than reported.

Code corrected:

In Version 4 both issues were fixed:

The non-compliant overrides of `maxDeposit` and `maxMint` were removed from the codebase. The base viewer functions implemented by Solmate's ERC-4626 are compliant with the standard.

The functions `maxWithdraw` and `maxRedeem` were modified to no longer cap the withdrawable assets by the current balance in the vault, and instead return the full amount. They still return 0 when total assets are zero, which is compliant since the `Vault` balance cannot decrease, so zero assets implies no shares outstanding.

6.18 Unused Variable

Informational **Version 2** **Code Corrected**

CS-INVMONO-031

The constructor of `Lender` stores the variable `minDebtFloor` as immutable, however it never uses it.

Code corrected:

The variable was removed in Version 4.

6.19 Incorrect Natspec Comments

Informational **Version 1** **Code Corrected**

CS-INVMONO-032

1. Code comment in `writeOff` imply that the collateral is sent to the caller, however it is sent to the specified `to` address. The caller is not the `to` address whenever `writeOff` is called directly (instead of during the liquidation).

```
// 3. send collateral to caller
collateral.safeTransfer(to, collateralAmount);
```

Version 2:

2. Code comments in `setMaxBorrowDeltaBps` imply that the max borrow delta is 5% (500 bps), however the actual limit must be between 200 bps (2%) and 50 bps (0.5%).

Code corrected:

The misleading comments were both corrected in Version 4.

6.20 Mismatch of Documentation and Code

Informational **Version 1** **Code Corrected**

CS-INVMONO-019

There are multiple discrepancies between the protocol's documented behavior in `README` and its implementation `Lender`. These mismatches particularly concern the handling of bad debt:

1. **Written-off Collateral:** The documentation claims remaining collateral is redistributed to borrowers who absorb bad debt. The code gives it to the `to` address specified by the `writeOff` caller.

2. **Write-Off Condition:** The documentation claims write-offs occur when `debt > collateral value`. The code requires `debt > collateral value * 100`.

Code corrected:

The documentation has been updated to accurately reflect the protocol's behavior as implemented in the codebase.

6.21 Missing Events for Admin Operations

Informational | Version 1 | Code Corrected

CS-INVMONO-020

The `Factory` contract does not emit events for critical admin operations: `setPendingOperator`, `setFeeRecipient`, and `setFeeBps`. Similarly, the constructor of the `Factory` contract emits no events.

In the Lender contract, `enableImmutabilityNow`, `pullLocalReserves` and `pullGlobalReserves` emit no events.

Emitting events for these operations would improve transparency and allow easier tracking of changes to key parameters and actions taken by the admin.

Code corrected:

Events have been added to the `Factory` and `Lender` contracts for the aforementioned admin operations.

6.22 Old Compiler Version

Informational | Version 1 | Code Corrected

CS-INVMONO-023

The Inverse Finance's contracts make use of an outdated compiler version *0.8.13* with some known vulnerabilities. It is considered a best practice to use the latest stable compiler version homogeneously throughout the project.

Code corrected:

The contracts have been updated to use Solidity version *0.8.24*.

6.23 Solmate Library Version and Deprecation

Informational | Version 1 | Code Corrected

CS-INVMONO-035

The protocol uses the Solmate library, which is not pinned to a specific tag or commit, but instead was copied directly into the repository. It is unclear what version was copied and if the version in scope has been audited.

For example, comparing the project's `SafeTransferLib` implementation to the latest version of Solmate reveals that it is missing some safety checks that were added in later versions.

Code corrected:

In Version 2, a recent commit of Solmate is pinned in a Git submodule. Note that Solmate is considered "soft deprecated" by its maintainers.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Liquidation Frontrunning

Informational | Version 3 | Acknowledged

CS-INVMONO-039

The liquidation function accepts a `minCollateralOut` parameter which acts as an "absolute" slippage protection. If the concrete liquidation amounts are not computed on-chain (i.e. the call is performed from an EOA rather than a smart contract), then the call is susceptible to front-running: if somebody front-runs the transaction with another liquidation, leaving less collateral than the specified `minCollateralOut` in the position, the "victim" transaction will revert. This creates a griefing vector that can potentially stall or slow down liquidations.
The "absolute" slippage protection is arguably unnecessary, because the code guarantees that the liquidator buys collateral at the right price (including the stipulated liquidation discount), whatever the concrete amounts end up being.

Acknowledged:

Inverse Finance has acknowledged the risk, but has decided to keep the code unchanged.

7.2 Gas Values Are Sensitive

Informational | Version 2 | Risk Accepted

CS-INVMONO-030

The Monolith uses hard-coded gas limits for `accrueInterest` and `writeOff` to prevent users from causing these functions to run out of gas and manipulate the system into triggering the catch state.

The provided values are 40,000 for `accrueInterest` and 120,000 for `writeOff`. These values are generally sensitive to current opcode costs and could change in future hard forks.

Furthermore, while `accrueInterest` is a pure function with known code, `writeOff` calls external contracts in unknown configurations (e.g., chained oracles or token contracts). This makes it difficult to statically estimate the required gas for this function.

In the testing configuration provided, `writeOff` consumed up to 93,000 gas, indicating that this threshold is more likely to be reached in practice.

Risk accepted:

Inverse Finance gave the following reasoning to hardcode the gas limit of `accrueInterest` to 40.000:

```
Given no external dependencies, this gas cost is unlikely to change.
If it is changed due to an Ethereum upgrade,
interest accrual will only be skipped in instances
```

```
where users only provide the minimum gas.
However, other users who provide sufficient gas
(higher than requirement) will continue gas accrual.
```

They decided on a threshold of 120.000 for `writeOff`:

```
Users are still able to provide more gas than required when calling write off.
Worst case scenario is that write offs may not be triggered at the end of liquidations
when users do not provide sufficient gas but can still be called by other users who provide more gas.
```

7.3 Fee Accumulators Can Overflow

Informational **Version 1** **Acknowledged**

CS-INVMONO-018

The protocol cast the `accruedLocalReserves` and `accruedGlobalReserves` as `uint120` to pack storage slots. While 120 bits are sufficient for regular amounts of tokens, they may not provide enough precision for tokens representing very low-value assets.

A 120-bit unsigned integer can represent numbers up to around 37 digits. At 18 decimals, this allows for representing amounts up to approximately 1e19 tokens. Very low-value `Coin` could in theory exceed that limit.

Acknowledged:

Inverse Finance has acknowledged the issue, but has decided to keep the code unchanged.

7.4 Initial Margin Formula

Informational **Version 1** **Acknowledged**

CS-INVMONO-033

The Monolith uses the same margin formula for creating positions (i.e. in `adjust`) as for liquidations (i.e. in `liquidate`): As such a user could easily create a position, wait for a price update and then immediately liquidate themselves for a profit. Similarly, they could open a position, `writeOff` a tiny amount of debt (1 wei of debt) and then immediately liquidate themselves.

Acknowledged:

Inverse Finance has acknowledged the issue, but has decided to keep the code unchanged.

7.5 Missing Sanity Check

Informational **Version 1** **Code Partially Corrected**

CS-INVMONO-021

1. The `minDebt` parameter is used to prevent the creation of tiny positions that would be uneconomical to liquidate. However, there is no sanity check that the value of `minDebt` during

deployment is greater than 1e18 (1 Coin). As such it is possible to set `minDebt` to a very low value, such as 0 which enables aforementioned attack vector.

2. The `psmAsset` should not be equal to `Coin` or `Collateral`. Similar `psmVault` should not be the same as `vault`. Otherwise, the internal accounting of the system could get broken. However, there are no sanity checks during deployment to prevent this.

Code partially corrected:

The `minDebt` parameter is now checked during deployment to ensure it is greater than a factory wide parameter `minDebtFloor`. This parameter is expected to be set to a significant value during construction. Further it is checked that the `psmAsset` is not equal to `Coin`, but it is not checked that `psmAsset` is not equal to `Collateral`.

7.6 Monetary Value of Dead Shares

Informational **Version 1** **Acknowledged**

CS-INVMONO-022

To ensure that inflation attacks are uneconomical, the first 10^{16} wei of shares deposited into the vault are burned. With stablecoin pegged to currencies such as the USD, EUR, CHF etc. the value is expected to be ~1 cent, which is negligible, but with currencies such as BTC, the amount could be upwards of 1,000 dollars which disincentivizes usage.

Additionally, the “dead shares” accrue yield, so the amount of inaccessible (stuck) funds grows over time.

Acknowledged:

Inverse Finance has acknowledged the risk, but has decided to keep the code unchanged.

7.7 Recovery of System After Shutdown

Informational **Version 1** **Acknowledged**

CS-INVMONO-034

The Monolith gradually decreases the collateral price to 0 once the Chainlink price feed has not been updated for 25 hours. However, if it starts reporting prices again it will start operating again as normal. During the shutdown period, no new borrowing can occur, but redemptions, liquidations and write-offs could create bad debt in the system. .. no state machine as liquity or others

---

Acknowledged:

Inverse Finance stated:

```
If there's bad debt after shutdown, we will add a UI warning
```

7.8 Return Value of Buy Fee Getter

Informational | Version 1 | Acknowledged

CS-INVMONO-024

The getter function `getBuyFeeBps()` returns 0 after the deadline is reached, even though buys are disabled after the deadline.

Acknowledged:

Inverse Finance:

```
Getters should not revert to avoid breaking integrators or frontends.
```

7.9 Unbounded Interest Rate Growth

Informational | Version 1 | Risk Accepted

CS-INVMONO-026

When the free debt ratio is lower than its target band, the interest rate will be increased exponentially. Since there is no upper bound, the rate could become very high and stay high until such a point that the free debt ratio goes above the upper end of the target band.

Note that more generally, the fact that the interest rate can grow, or decay exponentially means that, assuming the interest compounds, debt amounts can follow a double exponential since compounding interest itself is already an exponential function. Such behavior is not commonly seen in CDP or lending protocols and can be an excessive measure to maintain system health.

Risk accepted:

Inverse Finance has decided to not introduce an upper limit for interest rate growth. However, technical limits for the interest rate have been introduced to avoid overflowing the uint88 used to store the interest rate.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Invariant: Debt Share Prices Are Always at Least 1

Note Version 3

As of Version 3, after removing the "debt rescaling" feature, there is no more code path which simply decreases debt without also touching the shares. Instead, `totalPaidDebt` can increase by accruing interest, while `totalFreeDebt` can only increase through bad debt socialization, in `writeOff()`. Therefore, debt can increase by arbitrary amounts without changing the total shares.
Notice that both `increaseDebt()` and `decreaseDebt()` slightly decrease the share price, because of rounding errors when computing the shares delta.
Finally, notice that the share price starts off at 1.
We mathematically modelled this behaviour, and proved that, even though `increaseDebt()` and `decreaseDebt()` do decrease the share price, it cannot drop below 1.

8.2 Loose Access Control

Note Version 1

The system allows anyone to donate collateral or to repay debt on behalf of any other position, regardless of the delegation status set by the position's owner. Integrators have to take particular care, anticipating that their positions could change outside of their control.

8.3 Max LTV BPS

Note Version 1

The maximum LTV in basis points, which determines the LTV threshold for increasing user incentives, is set to the collateral factor plus a fixed 5%.

However, this fixed 5% addition leads to vastly different relative incentive thresholds depending on the base collateral factor. For example:

- With a **90% collateral factor**, the max LTV becomes 95%. This is a relative increase of only **5.5%** from the base LTV.
- With a **20% collateral factor**, the max LTV becomes 25%. This is a relative increase of **25%** from the base LTV.

8.4 No Auto Compounding of Interest

Note Version 1

The protocol does not compound interest automatically, but only when `accrueInterest()` is called. This means that the quoted APR can be misleading, since it can deviate from the realized yield (APY) depending on how often `accrueInterest()` is called. In particular, if `accrueInterest()` is only called once a year, then the APY equals the APR, but if it is called more frequently, then the APY is higher than the APR. The difference is small for common interest rates, but it can be significant for high interest rates. For example, at 40% APR, the APY is 46.41% if `accrueInterest()` is called quarterly and 49.15% if it is called daily.

8.5 PSM Behavior in Case of Depeg or Insolvency

Note **Version 1**

The Peg Stability Module can earn yield in two ways: either it can deposit its reserves into an ERC-4626 vault, or the reserve token itself can be rebasing. If ERC-4626 is used, if the vault makes a loss, the system will be insolvent, but users will still be able to sell at par assuming withdrawals are allowed by the vault. With some rebasing tokens, any depeg of the rebasing token will immediately affect the PSM exchange rate.

Take for instance Lido staked ETH: if an ETH-denominated Monolith instance is deployed using stETH as a PSM asset, the coin will tend to trade at the secondary market stETH:ETH exchange rate, which at times is not 1:1. This should not happen with AAVE deposit tokens, which can be instantly redeemed at 1:1 unless AAVE is experiencing a liquidity crisis, in which case withdrawals are blocked.

Stablecoin creators should take into account this behavior if they have the opportunity to use rebasing tokens, and should consider using a wrapper that protects against depegs if appropriate.

In the ERC-4626 vault case, `freePsmAsset` includes the principal amounts deposited and withdrawn and can therefore overestimate the assets in case the vault has generated a loss. In that case swaps can revert if they touch into the loss.

8.6 Price Feeds With Market Hours

Note **Version 1**

Price feeds for assets that are not traded 24/7 (e.g., stocks) can have market hours during which they are not updated. If such an asset is used as collateral, `Lender.getCollateralPrice()` would reduce the price to 0 within 49 hours of no updates.

Similarly, other assets could get updated, but would use the same price. One example is forex price feeds. These could not be supported by Monolith.

8.7 Redeemable Collateral Can Be Donated

Note **Version 1**

The Monolith does not track the total redeemable collateral in a internal state variable, instead it expects that the total redeemable collateral is the token balance of the `Lender` contract minus `nonRedeemableCollateral`. As such anyone can donate collateral to the `Lender` contract and increase the total redeemable collateral. This can be used to support the peg of the stablecoin as it collateralizes previously unbacked (redeemable) debt.

8.8 Redeemable Collateral Is Pooled

Note Version 1

The collateral of all depositors eligible for redemption is pooled. If the system has `totalFreeDebt > 0` but it is entirely bad debt (unbacked by collateral), then no redemptions can occur, and calls revert with "Insufficient redeemable collateral". However, once a new participant deposits collateral, their assets can be redeemed against existing bad debt. As such no-one should deposit collateral into the system and take out redeemable debt as long as there is (bad) redeemable debt in the system.

8.9 Redemption Can Revert Due to `minDebt` Check

Note Version 1

The `redeem` function requires that a borrower's remaining debt after a partial redemption is either zero or at least `minDebt`. When a position is healthy, its collateral value exceeds its debt, so the full debt can always be redeemed, leaving a remaining debt of zero. However, when the collateral cannot cover the full debt, a partial redemption may leave a remainder below `minDebt`, causing the transaction to revert.

Note, however, that since a redemption pays a premium on the collateral value, this scenario can only arise when the position is already liquidatable and is therefore unlikely to occur in practice.

8.10 Security Considerations for Bots

Note Version 1

The health of the system relies on keepers performing liquidations and redemptions. Developers of these bots should keep the following in mind.

1. Behavior of liquidate
    1. Repay amount is capped at 25% of total debt per call or 10,000, whichever is bigger. A bot may need multiple calls to fully liquidate a large position.
    2. Repay amount is further capped by the available collateral, so a `minCollateralOut` chosen for the original `repayAmount` may be too strict — a smaller liquidation can still be highly profitable.
    3. Collateral reward is capped by the user's collateral balance, and the liquidation incentive scales from 0% to 10%, so profit is non-linear with both position size and health.

The system also relies on redemption bots to restore the peg when the coin trades below its lower bound.

1. Behavior of redeem
    1. Redeem reverts when the remaining debt dips below minimum debt, so any redemption can be frontrun by another redemption or repayment that changes the borrower's debt.
    2. Redeemable amount is capped by the remaining debt and the max redeemable collateral, so a `minAmountOut` chosen for the original `amountIn` may be too strict — a smaller redemption can still be profitable.

Developers should be aware that data computed off-chain and passed to `redeem` or `liquidate` can be stale. They should use on-chain wrapper contracts that read live state and compute function arguments at execution time.

8.11 Security Considerations for Users

Note Version 1

Before interacting with the protocol, users should verify the following critical components to avoid immediate and total loss of funds:

1. **Collateral:** Ensure the collateral token is a legitimate and supported asset.
2. **PSM:** Verify that the PSM asset and its corresponding vault are legitimate and supported.
3. **Oracle:** Confirm that the oracle providing price feeds is reliable and correctly configured.
4. **System Health:** Review the total outstanding debt and the total supply of the protocol's native coin.
5. **Overall Backedness:** Assess the health of the system by checking the collateralization of other user positions.
6. **Debt Share Price:** Check that the current debt share price is not inflated.

Failure to verify these critical components can expose users to a malicious or misconfigured system, which could lead to an immediate loss of funds.

Additionally, users should consider the following non-exhaustive list of economic parameters to avoid unexpected losses over time:

1. **Half-Life:** Understand the protocol's half-life parameter, which is enforced to be between 12 hours and 30 days. This dictates the maximum speed at which interest rates can change.
2. **Interest Rate Parameters:** Review the interest rate parameters (e.g., target ratio, rate adjustment speed) to understand how the system responds to market conditions.
3. **Immutability Deadline:** Be aware of the immutability deadline, after which certain parameters can no longer be changed, and consider how this aligns with the protocol's trust model.
4. **Current Interest Rate and Recovery Time:** Be aware of the current interest rate. If the system has been dormant for a prolonged period and the interest rate has moved to an extreme value, the half-life parameter means it will take a significant amount of time for the rate to return to normal.

If these parameters are not well understood, they could lead to economic losses over time.

Version 2:

The Half-Life parameter has been updated to be between 24 hours and 30 days.

8.12 Threshold for Bad Debt

Note Version 1

When a position is liquidatable, its collateral is sold at a discount. If the health deteriorates so much that the (discounted) collateral value of the position no longer covers the debt, the position has caused bad debt to the system: this is because "maxing out" the liquidation will yield a "pure debt" position, with no collateral.
The threshold at which this happens depends on the liquidation discount: due to the specific behaviour of this system, namely that the liquidation bonus goes up from 0% to 10% as the position's LTV worsens

from `collateralFactor` to (`collateralFactor` + 5%), and that `collateralFactor` is capped at 85%, bad debt is incurred when the position's LTV reaches 90.9%.
This is a static threshold, that will be the same for all markets, and will not be configurable by deployers. Deployers should choose an appropriate `collateralFactor`, taking in consideration the fact that a liquidatable position causes no bad debt **if and only if** `collateralFactor < LTV < 90.9%`. The "window width" of (`90.9% - collateralFactor`) is therefore an important parameter to calibrate, as it is an indication of the system's resilience to sudden price fluctuations.

8.13 Unsupported Fee-on-Transfer Tokens

Note Version 1

The protocol does not correctly handle ERC-20 tokens that charge a fee on transfer. Several functions make calculations based on an expected transfer amount, rather than verifying the actual amount of tokens received:

1. Inbound Transfers: Risk of Insolvency

The most critical issue occurs when the protocol receives tokens. Functions like `Lender.adjust()` (when adding collateral) and `Lender.buy()` assume that the full amount specified in the function call is transferred to the contract.

However, with a fee-on-transfer token, the actual amount received is less than the expected amount. The protocol does not check its balance before and after the transfer to confirm the actual amount received. It proceeds with its internal accounting based on the *expected* amount, effectively crediting the user with more collateral than was actually provided. This discrepancy can be exploited to mint unbacked `Coin`, leading to insolvency.

2. Outbound Transfers: Incentive Issues

A less critical but still important issue occurs with outbound transfers. Functions such as `Lender.adjust()` (when removing collateral), `Lender.liquidate()`, `Lender.redeem()`, and `Lender.sell()` transfer tokens out to users. If a fee-on-transfer token is used, the user will receive less than the protocol intended to send.

While this would not directly harm the protocol's solvency, it can undermine economic incentives, particularly for liquidators who may receive a smaller-than-expected reward.

8.14 Vault Can Be Inflated

Note Version 1

The value of `Vault` shares can be inflated by sending assets directly to it. This will likely not be a profitable attack vector on vault depositors due to the existence of dead shares. However, other protocols built on top of the vault may be affected and could make the inflation attack on the vault profitable.

8.15 freePsmAssets Underestimate the Vault Position

Note Version 1

When the PSM is configured with an **ERC-4626 vault**, `freePsmAssets` only approximately tracks the `Lender's` vault balance.

In `sell()`, `freePsmAssets` is decremented by the full nominal `assetOut`, but both `convertToShares(assetOut)` and `redeem()` round down, so the vault gives away fewer assets than `assetOut`. Similarly, in `buy()`, `freePsmAssets` is incremented by `previewRedeemOrConvertToAssets(shares)` which rounds down but the asset increases by more.

These rounding errors will be given to local reserves when `accruePsmProfit()` is called.

8.16 getBuyAmountOut Does Not Account for Rounding

Note **Version 4**

The public viewer function `getBuyAmountOut(assetIn)` returns the amount of `Coin` a user would receive for a given `assetIn`. However, when a PSM vault is configured, the actual `buy()` function does not pass `assetIn` to `getBuyAmountOut`. Instead, it first deposits `assetIn` into the vault, then computes `redeemableFor = previewRedeemOrConvertToAssets(shares)` and calls `getBuyAmountOut(redeemableFor)`. Because of vault rounding, `redeemableFor` can be less than `assetIn`, so the actual `coinOut` may be lower than what `getBuyAmountOut(assetIn)` previews.

Off-chain integrators or contracts that call `getBuyAmountOut` to quote the expected output of a `buy` must be aware of this behavior to correctly estimate the amount of `Coin` received.