PUBLIC

Code Assessment

of the Grunt Smart Contracts

April 10, 2026

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 System Overview 7
4 Trust Model 13
5 Limitations and use of report 18
6 Terminology 19
7 Open Findings 20
8 Resolved Findings 31
9 Informational 75
10 Notes 81

1 Executive Summary

Dear 3F Team,

Thank you for trusting us to help 3F with this security audit. Our executive summary provides an overview of subjects covered in our audit. It refers to the latest reviewed state of Grunt according to the Scope. It aims to support you in forming an opinion on the security risks.

3F implements Grunt, a protocol enabling access to on-chain leveraged strategies with assets having asynchronous settlement.

The most critical subjects covered in our audit are escalation of privileges, correctness of integration with Morpho, and functional correctness. Security regarding all aforementioned subjects is high. Earlier versions of the protocol had a number of issues across these categories, all of which have been addressed:

On escalation of privileges, the protocol relies on several privileged accounts that are by design underconstrained in their operations to support a wide range of use cases. As these accounts are controlled by hot wallets, compromising them should not lead to a complete loss of user funds. Earlier iterations contained issues in which the operational freedom of these accounts could be abused to steal user assets (see Facilitator Steals From Other Intents via withdrawManager).

On the integration, the protocol integrates with Morpho Blue markets as a source of liquidity for the leveraged positions. Maintaining leverage requires the project to precompute Morpho state so that its own accounting remains consistent. We identified issues across three dimensions: A) calculations relied on stale market data (see Stale Morpho Market Data); B) calculations did not account for edge-case behavior of Morpho, such as health check during proportional collateral seizure (see Partial Liquidations Fail Health Check) or share price inflation (see Share Inflation via Direct Morpho Collateral Donation); C) rounding mismatches between the Grunt's calculations and Morpho (see Partial Pre-Liquidations Revert Due to Rounding Mismatch With Morpho).

In terms of functional correctness, earlier iterations contained issues such as incorrect debt accounting in share burning (see Burn Divides by Total Debt), NAV miscalculation due to bad-debt inclusion (see NAV Calculation Includes Bad Debt Positions), and a griefing vector that could permanently block repayment (see Permanent Blocking of Setting Status to Repaid).

The general subjects covered are documentation, trustworthiness, and integrations. Security regarding documentation is high, with extensive documentation available to explain code architecture and user flows. Security regarding trustworthiness is improvable, as key functionality of the protocol is heavily dependent on the liveness of privileged roles and their correct behavior. Users must trust these accounts to act dutifully and promptly if they want to invest into the leveraged positions. Further, any protocol integrating Grunt position manager shares must account for the possibility of share price inflation, see Share Price Inflation Possible Despite Mitigation.

In summary, we find that the codebase provides a good 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 services.

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	1
• Code Corrected	1
Medium-Severity Findings	16
• Code Corrected	16
Low-Severity Findings	48
• Code Corrected	33
• Risk Mitigated	3
• Specification Changed	1
• Code Partially Corrected	2
• Risk Accepted	6
• Acknowledged	3

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 Grunt 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	28 Jan 2026	fb03efa5fb6786b881aef017d10d926d19739a0c	Initial Version
2	13 Mar 2026	7056bb17257b7745fed054e7ba158f5f48cfda2c	Fixes
3	17 Mar 2026	8d0860fb8ca248787183092d29fba52f8c2df39e	Updated Fixes
4	31 Mar 2026	cb50ff25b9d608eef18df2cd2274b8fcea8c46ca	Updated Fixes
5	1 Apr 2026	bded62a403acf84f956258c0ebab61e5ed08d5b5	Updated Fixes
6	3 Apr 2026	12bd55140220e2f6f05353927ba6ba38646ef53f	Final Version

For the solidity smart contracts, the compiler version `0.8.34` was chosen.

The following contracts were in scope for the review:

```
request/abstract/tokens/ControlledToken.sol
request/abstract/tokens/TokenController.sol
request/abstract/vault/ControlledVault.sol
request/abstract/vault/VaultController.sol
request/abstract/OfferReceiver.sol
request/RequestFactory.sol
request/Vault.sol
request/Request.sol

facility/Facility.sol
facility/IntentDescriptor.sol

facility/base/FacilityFunds.sol
facility/base/FacilityIntents.sol
facility/base/FacilityLP.sol
facility/base/FacilityRequests.sol
facility/base/FacilityPositionManager.sol
facility/base/FacilityRoles.sol
facility/base/FacilitySwap.sol

funds/USCCFund.sol
funds/USCCFundFactory.sol
funds/WrappedAsset.sol

guard/TransferGuard.sol
guard/TransferGuardFactory.sol
```

```
borrow/MorphoBorrowPosition.sol
borrow/MorphoBorrowPositionFactory.sol

manager/base/PositionManagerBase.sol
manager/base/PositionManagerAdmin.sol
manager/base/PositionManagerLP.sol
manager/base/PositionManagerRebalancing.sol
manager/PositionManager.sol
manager/PositionManagerFactory.sol
manager/rebalancer/MorphoRebalancer.sol

libs/borrow/SharesMathLib.sol
libs/borrow/LibBorrowErrors.sol

libs/common/LibChecks.sol
libs/common/LibCommonErrors.sol
libs/common/LibPause.sol

libs/facility/LibAddress.sol
libs/facility/LibConstants.sol
libs/facility/LibFacilityErrors.sol
libs/facility/LibIntent.sol
libs/facility/LibStorage.sol
libs/facility/LibTokenBalances.sol

libs/funds/LibFundsErrors.sol
libs/funds/Order.sol

libs/manager/LibConstants.sol
libs/manager/LibExecutor.sol
libs/manager/LibManagerErrors.sol
libs/manager/LibOperations.sol
libs/manager/LibStorage.sol
libs/manager/LibView.sol

libs/request/Lib128Fields.sol
libs/request/LibAllowance.sol
libs/request/LibMintAuth.sol
libs/request/LibRequestErrors.sol
libs/request/LibTokenController.sol

libs/Constants.sol
```

In (Version 5) the following contracts were moved to another folder:

```
funds/USCC/USCCFund.sol
funds/USCC/USCCFundFactory.sol
```

2.1.1 Excluded from scope

Any contracts not explicitly mentioned in the scope (i.e., third-party libraries like Solady) are excluded from the scope. While the correct integration with third-party protocols like Morpho and Superstate is in scope, the protocols themselves are not.

The economic model of Grunt is also out of scope.

Further, the correct deployment configuration and role management are not in scope of this review.

3 System Overview

This system overview describes the Version 5 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.

3F offers Grunt, a protocol enabling leveraged strategies for protocols with asynchronous settlement (e.g. RWAs), where usual leverage strategies using flash loans are not possible. It does so via bridge loans: a Bridge Facilitator provides short-term capital that the Facility uses to reach the target position, with repayment enforced on-chain by Grunt.

The high-level flow is: LPs deposit their starting asset into an Intent (an on-chain work order to invest into or divest from a leveraged strategy); the Facilitator (an automated operator) draws bridge liquidity from a Request (the bridge loan contract), optionally routes capital through a Fund (an async wrapper for RWA-like protocols) and then interacts with the PositionManager (a leveraged borrow-position vault) that manages the leveraged positions. Finally it repays the Bridge Facilitator and resolves the Intent: LPs can then claim their pro-rata share of the resulting token balances.

3.1 Facility

The `Facility` is the central smart contract of the protocol. It holds all intent assets, enforces intent lifecycle transitions, and is the sole entry point for Facilitator operations. It is composed of several functional sub-modules: `FacilityIntents` (intent creation and configuration), `FacilityLP` (LP deposit, withdrawal, claim and revertDeposit), `FacilitySwap` (guardian-gated inter-intent assets swaps), `FacilityFunds` (Fund order execution), `FacilityRequests` (bridge loan drawdown and repayment), and `FacilityPositionManager` (PositionManager interactions). The Facility supports a global pause that halts all operations. This is strictly time-limited and must be initiated by specifying a fixed duration via `pauseFor(duration)`.

3.1.1 Intents

An `Intent` is a fully on-chain work order specifying what the Facility is permitted to do for a given strategy flow. An intent defines the `depositAsset` (what LPs provide), the `targetAsset` (the desired output asset), a `depositCap` (limiting deposits), a `guardKey` (a PositionManager address enforcing compliance), a `resolveStart` timestamp, a guardian signature `quorum`, and whether LP tokens are transferable (`transferableIntent`). Both `depositAsset` and `targetAsset` can independently be a `PositionManager`, which covers entry into, exit from, and migration between leveraged positions. When `transferableIntent` is false, LP token transfers between addresses are restricted.

Intents progress through three lifecycle states:

- **Depositing** (`resolveStart > block.timestamp`): LPs can deposit and withdraw 1:1, subject to the `depositCap`. The Facilitator can update the cap via `setDepositCap()` and can advance `resolveStart` to the current block via `lock()`, ending the depositing phase early.
- **Resolving** (`resolveStart ≤ block.timestamp`, not yet resolved): Deposits and withdrawals are closed. The Facilitator executes the configured strategy: drawing bridge loans and routing capital into or out of a PositionManager.
- **Resolved**: LP claims are enabled. Each LP receives a pro-rata share of all token balances held by the intent at resolution. An intent can only be resolved when there is no active Fund order and, if a Request is attached, the Request has been repaid. That ensures the bridge loan has been repaid.

Note that during resolution, the LP might receive `depositAsset`, `targetAsset` or any other asset. There is no guarantee that the `targetAsset` is received even in a successful flow. If the `depositAsset` was a `PositionManager` share and the `targetAsset` was USDC, the LP might also receive DAI.

3.1.2 Facilitator

The Facilitator is an operational role (typically an automated bot or keeper) that executes an intent. The Facilitator is authorised to: lock an intent early (advance `resolveStart` to the current block); advance the IFund state machine (`create`, `commit`, `unlock`, `recover`); pull bridge liquidity from an attached Request and repay it; interact with PositionManagers (`deposit`, `withdraw`, `burn`).

Furthermore, the Facilitator can attach or detach a Fund (`IFund`), attach or detach a Request, and perform intent-to-intent token swaps via `swap`. All of these specific operations require a quorum of EIP-712 guardian signatures. Note, that the quorum is allowed to be 0. In that case the Facilitator has unchecked power over an intent.

The Facilitator cannot unilaterally change an intent's core configuration parameters (deposit asset, target asset, guard key), resolve an intent while a fund order is pending or a Request remains unpaid, or move tokens to arbitrary addresses. All token movements are tracked in per-intent accounting, ensuring the final claim distribution reflects only what the intent actually holds.

3.2 Request

The `Request` contract is the on-chain settlement intermediary for bridge loans. Each Request is exclusive to a single intent and denominated in the PositionManager's debt asset. Prime Brokers provide liquidity by signing EIP-712 offers off-chain; each offer specifies a maximum principal (`ptAmount`), an expected return, an expiry, and a sequential nonce for replay protection and cancellation. Offers support both EOA and contract-based signers (EIP-1271). An authorised operator calls `consume()` to settle one or more offers: the Request validates the signature, then pulls the specified principal from the Bridge Facilitator into the Request itself. Offers support partial fills—the consumed `ptAmount` can be less than the offer's maximum, and the issued yield scales proportionally. As an alternative to signed offers, `authorizeMinting()` allows the owner to pre-approve a specific funder for a given PT/YT amount; the funder then calls `mint()` to deposit assets and receive tokens without needing a signed offer.

Upon consumption, the Request issues two token types:

- **PT (Principal Token)**: redeemable 1:1 for the funded principal; senior at redemption - PT holders are made whole before any yield is distributed.
- **YT (Yield Token)**: represents the agreed expected return, paid only after PT is fully covered. If repaid assets are insufficient, PT holders share the shortfall proportionally and YT receives nothing. YT holders can receive more than 1 token per unit of yield token. This is intended behavior in case of late repayments.

PT and YT are each implemented as ERC-4626 vaults with direct deposits disabled; only the Request contract (as `VaultController`) can mint shares. Redemption uses the standard ERC-4626 `redeem()` and `withdraw()` once the request is set to repaid.

If an offer includes a `useCallback` flag, the Request calls `onRequestConsumed()` on the maker's contract before pulling funds, allowing the Prime Brokers to source liquidity from internal positions and set allowances. Redemptions are unlocked either when `setRepaid()` is called by an authorised party, provided a mandatory cooldown period (`mintToRepaidDelay`) has elapsed since the last mint, or when the `repaymentDeadline` is reached - the deadline acts as a fallback, allowing PT/YT holders to redeem regardless of whether `setRepaid()` has been called. After the deadline is reached, no payments to the Request can be made.

3.3 Fund

The `IFund` interface defines a standardised state machine for wrapping protocols with asynchronous or multi-step settlement (RWAs, staking protocols, on-chain capital allocators). The states are:

- **EMPTY**: no order exists.
- **PENDING**: order submitted, awaiting pre-acceptance validation (e.g. KYC/AML checks); assets not yet transferred.
- **ACCEPTED**: order approved; Fund is ready to receive assets.
- **PROCESSING**: assets transferred; underlying operation in progress (deposit with external protocol, awaiting settlement period, etc.).
- **RECOVERING**: processing failed; assets held by Fund pending reclaim.
- **UNLOCKING**: processing complete; output assets ready to claim.
- **ENDED**: terminal state after `unlock()` or `recover()`.

The four primary operations are `create()` (submit an order—state becomes PENDING or ACCEPTED depending on whether the protocol requires pre-acceptance checks like KYC/AML), `commit()` (transfer assets to the protocol), `unlock()` (claim output on success), and `recover()` (reclaim input on failure). Each operation acts on an `Order` carrying an `input` and `output` amount. If the protocol settles at a different amount than requested, the operator calls `resolve()` to adjust the order before `unlock()` or `recover()` can proceed. The interface supports partial settlement: `unlock` and `recover` can return `PROCESSING` to signal remaining assets, allowing repeated calls until fully settled. An accepted or pending order can alternatively be cancelled to return the machine state back to **EMPTY**. Funds and Requests are exclusive to a single intent and cannot be reused. The only fund currently implemented is the **USCC Fund**: deposits transfer USDC to Superstate, receive USCC, and wrap it into wUSCC; redemptions burn wUSCC, unwrap to USCC, and redeem via Superstate to return USDC.

3.3.1 USCC Fund

Deposit: `commit()` transfers USDC to Superstate's `recipient` address under an off-chain agreement; Superstate mints USCC to the fund off-chain. Once the fund's USCC balance meets the expected output, `unlock()` wraps the received USCC into wUSCC and mints it to the Facility.

Redemption: `commit()` burns wUSCC (unwrapping to USCC held by the fund) and calls `offchainRedeem()` on the USCC token to trigger off-chain settlement with Superstate. Once Superstate returns USDC to the fund, `unlock()` transfers it to the Facility.

Recovery: `recovering()` transitions a pending order into the `RECOVERING` state, enabling the protocol to reclaim USDC (for deposits) or USCC (for redemptions) if off-chain settlement issues arise. Conversely, `cancelRecovering()` aborts a recovery attempt and reverts the transaction to the `PROCESSING` state, allowing the contract to resume waiting for Superstate's off-chain delivery.

3.3.2 Wrapped Assets

Receipt tokens received from an IFund (e.g. USCC from the USCC Fund) are not used as collateral directly. Instead, they are wrapped 1:1 into a controlled token (e.g. wUSCC) before being supplied to the PositionManager. The wrapper enforces transfer restrictions via three roles: `ISSUER_ROLE` (can mint and burn wrapper tokens), `SENDER_ROLE` (holder may send to any address), and `RECEIVER_ROLE` (holder may receive from any address). The only addresses granted `ISSUER_ROLE` are Fund contracts so they can mint wrapped tokens. Transfers between arbitrary addresses require the sender to hold `SENDER_ROLE` or the receiver to hold `RECEIVER_ROLE`; otherwise the transfer reverts. The Facility, PositionManager, MorphoBorrowPosition and Morpho markets are expected to hold the `SENDER_ROLE` to be able to send tokens. The Facility is expected to have the `RECEIVER_ROLE` to accept wUSCC as collateral. Further rebalancer contracts are expected to have the `RECEIVER_ROLE`.

The restrictions are in place to restrict who can supply collateral to Grunt's designated Morpho markets and by extension restrict users from borrowing on those markets, which would otherwise affect the functionality of the protocol.

3.4 Position Management

The `PositionManager` is an ERC-20 share vault aggregating one or more `IBorrowPosition` modules. Total assets equal the sum of each position's collateral value (quoted in the debt asset) minus outstanding debt, floored at zero so that positions with bad debt do not negatively impact the aggregate:

$$totalAssets = \sum \max(asset \cdot price - debt, 0)$$

Shares represent pro-rata net equity. Unlike ERC-4626, callers do not deposit a single asset; instead, `deposit(collateral, debt)` and `withdraw(collateral, debt, strategy)` mint or burn an `int256` share delta to the Facility reflecting the equity change. This allows the Facility to reconstruct the value held by each intent in the respective PositionManager.

Deposits and withdrawals are routed through ordered queues:

- **Supply queue**: used when increasing leverage. Each entry carries a per-entry borrow cap (`maxBorrow`). `deposit(collateral, debt)` iterates the queue, borrowing up to each venue's available liquidity and cap, and supplying the proportional collateral slice. It explicitly bounds leverage using `collateralForBorrow` and `borrowForCollateral` to ensure LTV limits are respected. If the requested debt cannot be fully sourced, the call reverts. Crucially, any remaining collateral not utilized for borrowing is deposited into the first position in the queue.
- **Withdrawal queue**: defines the ordered set of positions utilized when processing share redemptions. The protocol manages these outflows via a `WithdrawalStrategy`:
  - `withdrawSequential` drains positions one-by-one following the queue order. For each position, it attempts to fully repay outstanding debt and withdraw available collateral until the aggregate redemption target is met.
  - `withdrawProportional` executes a pro-rata distribution across all positions in the queue. It strictly apportions the debt repayment and collateral withdrawal based on each position's share of the total debt and collateral in the queue.

Burning shares is also done using `withdrawSequential` or `withdrawProportional`. When burning shares, the caller needs to repay the corresponding debt and receives the corresponding collateral. As such, burning shares should leave Position Health unchanged. As detailed below, exceptions apply. However, it is important to note that if a position was previously above the `PositionManager` LTV, then it will remain above the `PositionManager` LTV. As such, burning technically does not respect the `PositionManager` LTV.

Fees are computed on every share-minting operation: a management fee as an annualised rate on total assets (time-weighted), and a performance fee on gross gains net of management fees.

Fees are minted as PositionManager shares, diluting existing shareholders. It is important to note in case of oscillating `totalAsset` values and frequent calls to the PositionManager, performance fees will be charged multiple times, as they will be charged on every increase of `totalAsset`. This is the case even if the share price has not increased overall.

Due to the used formula, if a management fee of 2% is specified the fee recipient will, after fee share minting, not hold 2% of the shares, but less. This is expected.

The minting of the management fee shares slightly dilutes the performance fee shares as the performance fee calculation uses cached values for `totalSupply`. This is expected.

3.4.1 Morpho Borrow Position

`MorphoBorrowPosition` is the single `IBorrowPosition` adapter currently implemented. Each instance corresponds to one Morpho Blue market, identified by its loan asset, collateral asset, oracle, LLTV, and IRM parameters. The contract enforces a dual LTV system with invariant `0 < safeLtv < liquidationLtv ≤ marketLLTV`; both values are immutable after initialisation:

- **safeLtv**: the operational LTV ceiling enforced during `borrow()` and `withdrawCollateral()` operations. All operations that could worsen health are checked against this threshold.
- **liquidationLtv**: the threshold at which `preLiquidate()` becomes callable by any external liquidator.

`preLiquidate()` is a custom liquidation mechanism that targets a proportional exchange of collateral for repaid debt. To guarantee the remaining position returns to a healthy state, the seizable collateral and required repayment are strictly capped to satisfy post-liquidation LTV thresholds. The operation is atomic via Morpho's repay callback (`onMorphoRepay`): within the callback the contract withdraws collateral and transfers it to the liquidator, before Morpho pulls the repaid debt in the same transaction.

All `MorphoBorrowPosition` contracts of a `PositionManager` must reference Morpho markets that share the same oracle contract.

3.4.2 Rebalancing

A dedicated `REBALANCER_ROLE` may execute a sequence of low-level operations: `repay`, `withdrawCollateral`, `supplyCollateral`, `borrow` directly against registered `IBorrowPosition` modules without minting or burning shares. This allows redistribution of collateral and debt across venues (e.g. migrating between Morpho markets). Three safety constraints apply: only modules already registered in the PositionManager may be targeted; a rebalance cooldown period is enforced between operations; and the PositionManager enforces a configurable `maxRebalanceLoss` (in basis points) by comparing total assets before and after the rebalance sequence, reverting if the loss exceeds the threshold. Any residual collateral or debt tokens remaining after the sequence are forwarded to a caller-specified receiver.

The `MorphoRebalancer` is a standalone contract that holds the `REBALANCER_ROLE` and sources debt tokens via Morpho flash loans. It requires `collateral = 0` in the rebalancing data as only debt tokens are flash-loaned.

3.5 Guard

The `TransferGuard` contract provides compliance-level validation for share token transfers. It is used by the `PositionManager` (for ERC-20 share transfers) and optionally by intent LP tokens. Each token is configured independently in one of two modes:

- **Blocklist** (default): all transfers are permitted except those involving addresses with `BLOCKLIST` status.
- **Whitelist**: only addresses with `WHITELIST` status may send or receive tokens.

`canTransfer(token, from, to, amount)` is called by the token contract on every transfer and reverts if the transfer is not permitted. Transfers can also be halted by a per-token pause, controlled by the `PAUSER_ROLE`. Address statuses are managed by the `COMPLIANCE_ROLE` and can be updated in batch.

3.6 Changes in Version 2

Version 2 introduces enhancements aimed at enforcing stricter validation in administrative functions and incorporating LTV constraints more comprehensively. In addition to these safety improvements, various logic and accounting bugs were resolved. The most notable changes across Grunt's components are highlighted below.

3.6.1 Facility

- **Guardian Signatures**: `setFund` and `setRequest` now require EIP-712 guardian signatures, preventing the Facilitator from unilaterally changing these critical components.
- **Compliance Reversals**: A new `revertDeposit` function allows the `Owner` or `COMPLIANCE_ROLE` to force-withdraw a user's full deposit. This operation is permitted even during the resolving phase.

3.6.2 Request

- **Repayment Constraints**: A strict 90-day maximum limit was added to the repayment deadline to prevent excessive lockups. Additionally, `setRepaid` now requires a minimum balance and enforces a delay (`mintToRepaidDelay`) after the last minting operation.
- **Slippage Protection**: The `mint` function now includes `minPt` and `minYt` parameters.

3.6.3 Fund

- **Liveness**: A `cancelRecovering` function was added to prevent permanent state deadlocks, allowing the protocol to exit the `RECOVERING` state if necessary.

3.6.4 Position Management

- **NAV Calculation**: The `totalAssets` formula now excludes bad debt positions.
- **Administrative Checks**: `addBorrowModule` now validates asset compatibility, contract ownership, and LTV alignment. `setLtv` ensures all active modules meet the new LTV threshold. Supply and withdrawal queues now enforce entry uniqueness.
- **LTV-Aware Deposits**: `processDeposit` now bounds leverage using `collateralForBorrow` and `borrowForCollateral`.
- **Withdrawal Refactoring**: Consolidated `processBurn` into `processWithdrawal` via `WithdrawalStrategy`. `withdrawSequential` drains positions greedily following the queue, while `withdrawProportional` distributes repayments and withdrawals pro-rata based on each position's collateral and debt.
- **Share Settlement**: `_settleShares` now uses conservative rounding (up for burns) and enforces pause checks on unchanged assets.
- **Fee Logic**: Maximum management fees are capped at 2% (reduced from 50%). Performance fees are now net of management fees.
- **Rebalancing**: A rebalance cooldown check and a strict `maxRebalanceLoss` cap were added to prevent a compromised Rebalancer from moving or losing all funds in a single transaction.

3.6.5 Morpho Borrow Position

- **State Freshness**: Relevant operations now call Morpho's `accrueInterest` to ensure up-to-date market data.

- **Liquidation Safety**: The new `_computeSeizedAsset` and `_computeRepaidShares` functions factor in LTV to calculate the maximum seizable asset and required repaid shares, ensuring the position LTV becomes safe after `preLiquidate`.

Version 2 also introduced deployment tracking across all factories, and migrated storage to ERC-7201 namespaced layouts for better upgradeability.

3.7 Changes in Version 4

- **Repayment Front-running Protection**: `setRepaid` accepts an additional `maxBalance` parameter. Together with the existing `minBalance`, it bounds the acceptable asset balance of the Request at the time of repayment, preventing a facilitator from inflating PT / YT value at the expense of intent depositors or deflating it at the expense of bridge facilitators.
- **Mint Slippage Protection**: `mint` replaces `minPt` with a `maxPt` parameter, capping the amount of token pulled from the caller. This prevents a consumer from front-running the `mint` transaction by increasing the caller's PT mint authorization to force a larger deposit than expected.

3.8 Changes in Version 5

- **Prime Broker → Bridge Facilitator**: The "Prime Broker" term has been renamed to "Bridge Facilitator" throughout the documentation and codebase. The role and functionality remain unchanged.
- **Stale Order Sync**: `resolve()` and `setFund()` now call `syncEndedOrder` before their main logic. When a Fund externally ends an order (e.g. via `forceEnd()`), this function detects the `ENDED` state and automatically deletes the intent's order and fund so the intent can proceed..

4 Trust Model

The protocol defines three trust levels for roles and external dependencies:

- **Fully trusted**: A compromised holder of this role can, in the worst case, steal or permanently destroy all user funds through direct administrative action or a combination of parameter changes and contract upgrades. No technical guardrail prevents this.
- **Semi-trusted**: A compromised holder can cause partial fund loss or liveness failure, but by design should not be able to cause a complete loss of user funds through their designated function set alone.
- **Untrusted**: External counterparties whose behaviour is constrained and validated entirely on-chain by the protocol's contract logic.

All major contracts (`Request`, `PositionManager`, `MorphoBorrowPosition`, `USCCFund`, `TransferGuard`) are deployed behind beacon proxies. Each factory owns an `UpgradeableBeacon` whose implementation pointer is shared by all proxy instances it created. A beacon owner can upgrade every deployed instance atomically in a single transaction, replacing contract logic for all users simultaneously.

4.1 Bots

The Facilitator, Rebalancer, Curator and Consumer are automated operators (typically backend bots or keeper scripts) and are classified as **semi-trusted**. If compromised or acting maliciously, they can cause partial fund losses or halt key protocol operations. By design, a **compromised bot should not be able to cause a complete loss of funds for users or prime brokers**.

4.1.1 Facilitator

The Facilitator is a semi-trusted role executed by a backend bot that drives intent lifecycle transitions inside the Facility.

In terms of liveness, the Facilitator is trusted to be available. If the Facilitator were inactive for a prolonged period, the following could occur:

1. Bridge loan funds are never pulled from an attached Request, or if pulled, are never repaid. Bridge Facilitators would be left waiting until the `repaymentDeadline` enables redemption as a fallback.
2. Assets remain stuck in Fund contracts, unable to progress through the settlement state machine.
3. No intent is created, existing intents cannot advance to resolving or resolved. As such no assets are deployed and no LP claims are possible (as such PM shares are essentially worthless).
4. No swap between different intents can happen.
5. No assets can be deposited or withdrawn from PositionManagers. Assets will continue paying fees.

In terms of safety, the Facilitator is trusted to choose correct parameters. If mistakes occur or the Facilitator is compromised, the following could happen:

1. The Facilitator deliberately delays deploying capital, allowing elapsed time to inflate management and performance fees at LP expense.
2. The Facilitator takes too much debt for an intent. Thereby, they are increasing the interest payments and modifying the leverage.
3. The Facilitator can re-leverage a PositionManager however they like. Generally, a PositionManager should keep a certain leverage, but a Facilitator can bring a 3x PositionManager to a 20x or a 1x leverage. This could lead to risky scenarios and might sometimes be tricky to undo.
4. If there were ever stuck tokens in one of the funds then these could get stolen by a malicious facilitator by creating a malicious intent with a quorum of 0 then withdrawing them with a malicious order.

4.1.2 Rebalancer

The Rebalancer is a semi-trusted role authorised to rebalance collateral and debt in the registered `IBorrowPosition` modules.

Two on-chain safety constraints bound the Rebalancer's power: only modules already registered in the PositionManager's `borrowModules` set may be targeted (unregistered positions revert with `UnauthorizedPosition()`); and a configurable `maxRebalanceLoss` threshold causes the transaction to revert if total assets decrease by more than the permitted basis-point loss after the rebalance sequence completes.

If the Rebalancer were compromised it could: migrate positions to Morpho markets with higher interest rates, increasing the cost of the leveraged strategy; reduce position health to the `safeLtv` threshold in markets and generate a loss up to the `maxRebalanceLoss` threshold.

Further, the `Rebalancer` can migrate debt and collateral between modules as long as each resulting position respects the `safeLtv`. When not all modules are included in the supply or withdrawal queue, the Rebalancer could shift debt into queued modules, making them less collateralized and causing user deposits or withdrawals through those modules to revert.

A compromised Rebalancer can also re-leverage a PositionManager however they like. Generally, a PositionManager should keep a certain leverage, but a Rebalancer can bring a 3x PositionManager to a 20x or a 1x leverage. This could lead to risky scenarios and might sometimes be tricky to undo.

Lastly, the PositionManager is not meant to permanently hold tokens as its funds are in the positions. However, in case the PositionManager holds any tokens, e.g. because dust has accumulated, the

Rebalancer can sweep those tokens out of the PositionManager. They are not part of the loss calculation.

Important: The `PositionManager` admin can set the rebalance cooldown to a very small duration (i.e., 0 seconds). In this special case, the rebalancer must be considered fully trusted.

4.1.3 Curator

The Curator is a semi-trusted role authorised to set the PositionManager's supply queue (an ordered list of `IBorrowPosition` modules with per-entry borrow caps used when levering up) and withdrawal queue (an ordered list of modules used when deleveraging). Only modules registered in `borrowModules` by the Owner may appear in either queue.

If the Curator were compromised it could: set the withdrawal queue to an empty list, which would revert all `withdraw()` and `burn()` calls, blocking users from exiting their positions; set the supply queue to exclude all markets, preventing new capital deployment; or route operations to spike interest rates.

If the Curator is not reacting quickly enough to market changes, they might not adjust `maxBorrow` values in the supply queue accordingly which blocks the Facility from depositing into the PositionManager.

4.1.4 Consumer

The Consumer is a semi-trusted role on the `Request` contract, intended to be held by a backend service, that coordinates which prime brokers will provide funds to a particular `Request`.

If the Consumer were compromised it could:

1. When calling `consume()` on a `Request`, the Consumer submits a `ptAmount` significantly smaller than the offer's signed maximum. This could essentially consume all signed offers by prime brokers without collecting significant funds. Thereby, it would delay the bridge loan process.
2. Allow any account to act as a bridge facilitator.

4.2 Owners and Admins

Owners and Proxy Admins are **fully trusted**.

The beacon proxy ownership is the single highest-impact privilege in the system: a single transaction by the beacon owner replaces the logic of all deployed proxy instances for all users simultaneously. Similarly the proxy owner of the Facility is equally fully trusted.

Additional owner capabilities by contract:

- **Facility Owner**: Creates intents with arbitrary parameters and updates `targetAsset` on depositing intents.
- **PositionManager Owner**: Adds and removes borrow modules; sets management and performance fees (management fee is capped at 2% and performance fee at 50% by the contract); sets the operational `safeLtv` and `liquidationLtv`; sets `maxRebalanceLoss`; attaches or removes the `TransferGuard` (setting it to `address(0)` removes all transfer restrictions on PositionManager shares).

  Currently, the `MorphoBorrowPosition` is the only supported borrow module. Future borrow module must not allow any reentrancy possibility in the calls specified by `IBorrowPosition`.

  When adding borrow modules they need to check the following on the Morpho Market:

  - Has a correct oracle (provides correct price with correct decimals, does not revert)
  - Has the correct Interest Rate Model
  - LLTV is as expected

- There is no bad debt
- The borrow share price has not been significantly inflated or deflated
- No other borrow modules of the `PositionManager` references the same Morpho Market.

- **TransferGuard Owner**: Configures whitelist or blocklist mode per registered token; grants and revokes the `COMPLIANCE_ROLE` and `PAUSER_ROLE`; can freeze all transfers for any registered token or remove all restrictions. Can essentially break the system by blocklisting crucial contracts.
- **WrappedAsset Owner**: Grants and revokes `ISSUER_ROLE` (mint/burn wUSCC), `SENDER_ROLE` (authorise sending wUSCC to any address), and `RECEIVER_ROLE` (authorise receiving wUSCC from any address), controlling who may supply the wrapped collateral to Grunt's designated Morpho markets. Removing the `SENDER_ROLE` from the Morpho Market would stop liquidations.
- **Request Owner**: Marks the request as repaid via `setRepaid()` (enabling PT/YT holder withdrawals), consumes signed offers and authorises minting for specific addresses (shared with the `CONSUMER_ROLE`), adjusts the `mintToRepaidDelay`, and grants or revokes the `PULLER_ROLE` and `CONSUMER_ROLE`. Critically, here the owner needs to protect bridge facilitators and intent shareholders. Note that it is fully trusted for `Request` contracts that have been properly checked by the Guardians as part of `setRequest`.

These roles are expected to be held by the protocol team behind a multisig or governed timelock. No technical guardrail enforces this requirement or delays the execution of owner transactions.

4.3 Other

Facility Pauser: Holds the `COMPLIANCE_ROLE` on the Facility and may call `pauseFor(duration)` to halt all Facility operations. If compromised, the Pauser can freeze deposits, withdrawals, claims, and all Facilitator actions for the duration of the pause, creating a liveness failure.

TransferGuard Pauser: Holds the `PAUSER_ROLE` on the TransferGuard and may call `pause()` / `pauseFor(duration)` to halt selected token operations. This can be used to prevent rebalancing of a PositionManager, which could lead to liquidations if rebalancing is prevented for a longer time.

TransferGuard and Facility Compliance: Holds the `COMPLIANCE_ROLE` on the TransferGuard and may add or remove individual addresses from the blocklist (or whitelist, depending on mode). They can also call `revertDeposit` on the Facility to force-withdraw a user's full deposit. If compromised, the Compliance role could blocklist LP holders or PositionManager participants, preventing them from transferring tokens and therefore blocking key functionality, or arbitrarily revert users' deposits. A compromised Compliance role could also allow anyone to hold restricted assets or blocklist crucial system contracts and thereby temporarily DoS the system. If the `Facility` were blocklisted, any interaction with the `PositionManager` that involves minting or burning shares would be blocked.

Guardian: Guardians are fully-trusted signers who produce EIP-712 signatures authorising inter-intent token swaps, as well as `setRequest` and `setFund` operations submitted by the Facilitator. Each intent defines a `quorum`—the minimum number of guardian signatures required for these calls to be accepted. Guardians cannot initiate operations independently; their role is limited to signing authorisations. If all guardians are unavailable, these operations are blocked (liveness risk) but no fund loss follows directly. A malicious guardian can sign arbitrary operations, and could steal user funds when colluding with the Facilitator.

Operator: The Operator is a fully-trusted role in the `USCCFund` that triggers the recovery flow. If compromised, the Operator could call `setOracle` to set a malicious oracle, manipulating the return value of `totalAssets()` and rendering the slippage check ineffective. The Operator can also directly overwrite the input and output amounts of an order, bypassing the slippage check entirely.

Deployer: The deployer must correctly initialise all contracts, assign roles to the intended addresses, and not retain privileged roles (such as Facilitator or Operator) after deployment unless that is the explicit operational design.

Superstate: Fully trusted to faithfully mint USCC upon receiving USDC and to process off-chain redemption requests. If Superstate fails to mint USCC after a `commit()` call, the fund order remains in PROCESSING state indefinitely until the Operator calls `recovering()` and Superstate returns the original USDC. If Superstate's allowlist rejects the USCCFund contract address, `create()` will revert, preventing any new deposit orders. Superstate is trusted to always fill orders completely.

Chainlink: Fully trusted to return a current USCC/USD NAV price. The most critical consumer is Morpho Blue: the oracle is used directly by Morpho to determine collateral value for every borrow health check and liquidation decision. An incorrect or stale price could allow undercollateralised borrows, prevent valid liquidations, or trigger unwarranted ones. The PositionManager also relies on this price to compute `totalCollateralQuoted` and `totalAssets`; a bad price would distort share accounting and fee accrual.

USDC: Fully trusted. Oracle assumes that 1 USDC is always worth 1 USD.

Prime Brokers and LPs: Untrusted.

Morpho: Morpho is trusted not to allow any reentrancies. Hence, the Interest Rate Model should not make calls to untrusted code.

Facility: There are multiple roles that are intended for the Facility, e.g. `PULLER_ROLE` in `Request`, `MINTER_ROLE` in `PositionManager` or `_DEPOSITOR_ROLE` in `USCCFund`. We assume that only the Facility is holding these roles.

4.4 Supported ERC-20

The system handles different ERC-20 tokens as deposit, debt, collateral or withdrawal assets. The following ERC-20 tokens are **not** supported:

- Any type of reentrant token (e.g., ERC-777 or other tokens with transfer callbacks).
- ERC-20 where one wei (smallest unit) has a value of more than 0.0001 USD.
- ERC-20 where one wei (smallest unit) has a value of less than `10**-25` USD.
- ERC-20 with Fee on Transfer.
- Rebasing ERC-20 where the balance can change without transfers.
- ERC-20 that revert on very large approvals
- ERC-20 with more than 18 decimals
- ERC-20 with multiple entrypoints

4.5 Miscellaneous

The protocol is designed for deployment on Ethereum Mainnet. Before deploying to a different chain, a separate assessment of chain-specific differences and their impact on Grunt functionality must be performed.

5 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.

6 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.

7 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
- Trust: Violations to the least privilege principle

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	11

- Behavior of PM Deposits, Withdrawals and Burns With Bad Debt Risk Accepted
- Facility Allows Payable ERC-6909 Transfers Risk Accepted
- Last Shareholder Cannot Fully Exit the PositionManager Acknowledged
- Management Fees Incorrectly Estimated Risk Accepted
- Missing Funding-to-Utilization Phase Boundary in Request Lifecycle Acknowledged
- Missing Token Allowlist Risk Accepted
- Operator Can Cause Loss of Funds for a Particular Intent Risk Accepted
- Share Inflation via Direct Morpho Collateral Donation Code Partially Corrected
- USCCFund Is Missing Chainlink Price Freshness Check Risk Accepted
- USCCFund's totalAssets Reports Global wUSCC Supply Instead of Per-Fund Holdings Acknowledged
- Unbacked Yield Token Minting Leads to Yield Theft Code Partially Corrected Risk Accepted

7.1 Behavior of PM Deposits, Withdrawals and Burns With Bad Debt

Design Low Version 2 Risk Accepted

CS-GRUNT-084

When the `PositionManager` computes `totalAssets`, `LibView.totalAssets()` sums per-position asset value as `collateral.zeroFloorSub(debt)`, flooring bad-debt positions to zero instead of letting them reduce the total asset value. However, state-modifying functions are not consistent with this behavior. Consider the following two examples:

1. As part of a **withdraw** operation: `LibOperations._withdrawSequential()` iterates the queue, repaying debt before withdrawing collateral. When the queue starts with a bad-debt position, repaying its debt does not change `totalAssets` (the position is already floored to zero),

yet withdrawing collateral from subsequent healthy positions does. Because `PositionManagerLP._settleShares()` prices burned shares against the `totalAssets` delta, the withdrawer burns more shares than the value received. Therefore, the withdrawer could lose significant funds.

2. As part of a **deposit** operation: `processDeposit()` first deposits collateral and then borrows debt. Depositing collateral into bad debt modules does not increase `totalAssets`, while borrowing from collateralized modules decreases `totalAsset`. So similar to **withdraw**, `_settleShares()` mints fewer shares than the value deposited absorbing some of the bad debt and the caller loses funds.
3. As part of a **burn** operation: `PositionManagerLP.burn()` computes pro-rata collateral and debt from `totalCollateral` and `totalDebt`. If that strategy is chosen, it then passes these values to `LibOperations._withdrawSequential()`. Bad debt will be repaid, not increasing `totalAssets`. Then collateral will be taken from overly healthy positions. Because `totalDebt` includes the bad debt, the caller repays more value than they receive, absorbing part of the bad debt. The loss of the caller depends on the size of the bad debt relative to the `totalAssets`.

   Additionally, the share price will drop as `totalAssets` will decrease. This violates an important invariant, namely that burning shares should not significantly influence the share price.

A compromised Curator could exploit this behavior by reordering the withdrawal queue to place bad-debt modules first, causing subsequent `deposit`, `withdraw`, and `burn` operations to silently absorb bad debt at the expense of callers and generally LPs.

Risk accepted:

3F accepted the risk of bad-debt absorption during deposits, withdrawals, and burns and provided the following reasoning to keep the code unchanged:

> One approach considered was to skip positions carrying bad debt during user interactions, for example in _withdrawSequential(), and rely on rebalancing to resolve the bad debt since it is already socialized in the share price.
>
> However, this would require a substantial change and is not aligned with how this situation will be handled in practice. Instead, the strategy is to effectively abandon PositionManager instances that reach this state, rather than continuing to support user interactions on a PositionManager affected by bad debt.

7.2 Facility Allows Payable ERC-6909 Transfers

Design | Low | Version 1 | Risk Accepted

CS-GRUNT-016

The `Facility` contract overrides ERC-6909's `transfer` and `transferFrom` to block transfers to `address(0)`, preserving correct `totalSupply` tracking. These overrides inherit the `payable` modifier from Solady's ERC6909 base implementation, which uses it as a gas optimization.

However, the Facility contract has no reason to accept ETH during share transfers. There is no logic that handles or accounts for `msg.value`. The `payable` modifier is carried over solely because the parent signatures require it for the override to compile, not because the Facility needs it.

Users or integrating contracts could accidentally send ETH with a transfer call, permanently locking those funds in the Facility contract with no recovery path. Removing `payable` enforces that no ETH accompanies share transfers, eliminating a class of user-error fund loss.

Risk accepted:

3F stated:

```
We don't want to override the ERC-6909 transfer functions and we accept the consequences.
```

7.3 Last Shareholder Cannot Fully Exit the PositionManager

Design Low Version 1 Acknowledged

CS-GRUNT-064

When the last remaining shareholder attempts to redeem all their shares and recover all assets from the `PositionManager`, neither `withdraw` nor `burn` reliably achieves a clean exit.

The `withdraw` function requires the caller to specify an exact `debt` amount at transaction creation time. Because Morpho debt accrues interest continuously, the actual debt at execution time will differ from the value specified in the transaction. This causes the transaction to either revert with `ExcessDebtRepay` (if debt grew) or leave behind unrepaid dust (if less debt is repaid than exists).

The `burn` function calculates debt to repay per position via

```
debtToRepay.mulDiv(positionDebt, totalDebt)
```

in `processBurn`, which rounds down. When there are multiple borrow positions, each per-position repayment is rounded down independently, leaving behind small dust amounts of debt across positions. This residual debt prevents the corresponding collateral from being fully withdrawn, as Morpho's health check would fail. As a result, the last shareholder either cannot exit cleanly or leaves dust amounts of collateral and debt locked in the contract.

Acknowledged:

3F stated:

```
Acknowledged. The residual dust is not a significant problem in practice.
```

7.4 Management Fees Incorrectly Estimated

Design Low Version 1 Risk Accepted

CS-GRUNT-022

The `_accrueFees()` function in `PositionManagerBase` computes the management fee by applying the annual rate to `currentTotalAssets` (the live `totalAssets()` value at the moment of accrual) multiplied by the elapsed time since the last accrual. The formula is:

```
managementFeeAssets =
   currentTotalAssets * managementFee * elapsed / (BPS * SECONDS_PER_YEAR)
```

This treats `currentTotalAssets` as if it had been at that level for the entire elapsed period, when in reality the assets may have changed over that window.

If `totalAssets` was 1M at the last accrual and grew to 2M by the next, the fee is charged as if 2M had been under management for the full duration. The overcharge dilutes LP holders in favor of the fee recipient on every accrual where assets have increased.

Similarly, if `totalAssets` decreases, management fees would be underestimated.

The magnitude scales with both the change and the time between accruals: infrequent accruals during periods of asset change (e.g., rising collateral prices) produce the largest errors. In practice, if accruals happen frequently the bias is small, but there is no guarantee of frequent accruals and no upper bound on the elapsed time.

Risk accepted:

3F stated:

```
We don't know what the AUM curve will look like; we take the simplest
approach for fee estimation.
```

7.5 Missing Funding-to-Utilization Phase Boundary in Request Lifecycle

Correctness | Low | Version 1 | Acknowledged

CS-GRUNT-059

The README documents a three-phase lifecycle for `Request`: **Funding**, where depositors call `consume()` or `mint()` to supply assets in exchange for PT/YT tokens; **Utilization**, where the puller calls `pullFunds()` to deploy capital and later `repay()`; and **Redemption**, where `setRepaid()` enables PT/YT holders to redeem.

The `Request` contract only tracks a single `bool repaid` flag in `RequestStorage`. So **Funding** and **Utilization** are effectively combined with no state transition taking place on `pullFunds()`. Consequently, `mint()` and `consume()` remain callable during the Utilization phase.

Acknowledged:

3F stated:

```
A mint-to-repaid deadline was added, enforcing a minimum time before setting
as repaid after consuming funds. The state transition is intentionally
non-strict to allow increasing the borrow amount in case of errors and to
avoid wasting a request for funds.
```

7.6 Missing Token Allowlist

Security | Low | Version 1 | Risk Accepted

CS-GRUNT-065

The system does not enforce an allowlist for tokens that can be used in intents. Through swaps, requests, and position managers, arbitrary tokens can be added to an intent via token snapshots. These could be non-standard or malicious ERC-20 implementations that are incompatible with the protocol's accounting system.

A compromised facilitator could exploit this by introducing an unsupported token to break system accounting. If an intent holds a token that reverts on `transferFrom`, no user can ever claim or withdraw their funds.

Risk accepted:

3F stated:

```
We do not support exotic ERC20s.
```

7.7 Operator Can Cause Loss of Funds for a Particular Intent

Trust | Low | Version 1 | Risk Accepted

CS-GRUNT-026

In `USCCFund`, the `resolve()` function allows privileged roles (`OWNER` and `OPERATOR`) to overwrite the `input` and `output` of the associated order. By setting either of `input` or `output` to 0, the expected state transition checks are bypassed. Therefore `recover` and `unlock` immediately succeed even though the expected funds might not have arrived.

If the `Facilitator` automatically calls `recover` or `unlock` when it observes that they are callable, the associated intent loses all its funds.

```
function resolve(Order memory order, uint256 input, uint256 output)
                        external onlyOwnerOrRoles(OPERATOR_ROLE) {
    ...

    order.input = input;
    order.output = output;

    _storage.resolvedOrder = order;

    ...
}
```

However, the Operator cannot steal these funds without help from another privileged role.

Risk accepted:

3F has accepted the risk, but has decided to keep the code unchanged. They provided the following reasoning:

```
In the case of USCCFund , the operator will be a multisig.
The flexibility might be important in some situations.
```

7.8 Share Inflation via Direct Morpho Collateral Donation

Security | Low | Version 1 | Code Partially Corrected

CS-GRUNT-032

The `PositionManager` computes `totalAssets()` as oracle-quoted collateral minus debt across all its borrow modules. Shares are minted via `_settleShares()` using a virtual-offset formula:

```
function convertToShares(uint256 assets, uint256 _totalSupply, uint256 _totalAssets)
  internal pure returns (uint256 shares)
{
  return assets.mulDiv(_totalSupply + VIRTUAL_SHARES, _totalAssets + VIRTUAL_ASSETS);
}
```

Where `VIRTUAL_SHARES = 1e6` and `VIRTUAL_ASSETS = 1`. This offset is modeled after Morpho's inflation-attack mitigation.

However, anyone can inflate `totalAssets` without minting shares by calling `morpho.supplyCollateral(marketParams, amount, borrowPositionAddress, "")` directly on Morpho Blue. This function has no authorization — it accepts any `msg.sender` supplying collateral on behalf of any address, including the PositionManager's borrow module.

This donation increases the borrow position's collateral (and thus the PM's `totalAssets()`) without going through the PM's `deposit()` flow, creating an asymmetry between assets and shares.

7.8.1 Example Attack Flow

Step 0 — Initial state.

The `PositionManager` has a configured borrow module (`MorphoBorrowPosition`) but no depositors yet.

- `totalSupply = 0`
- `totalAssets = 0`

Step 1 — Donation.

The attacker calls:

`morpho.supplyCollateral(marketParams, D, borrowPositionAddress, "")`

This is called directly on Morpho, donating `D` wei of collateral. This bypasses the PM entirely. No shares are minted.

- `totalSupply = 0`
- `totalAssets = D`

Step 2 — Victim deposits.

The victim calls `deposit(V, 0)` through the Facility. Inside the `PositionManager`, `_settleShares` computes:

```
sharesToMint = V * (0 + 1e6) / (D + 1) = V * 1e6 / (D + 1)   [floor division]
```

The division rounds down. The victim receives fewer shares than the fair value of their deposit.

Step 3 — Attacker deposits.

The attacker calls `deposit(A, 0)` with `A` chosen so that `A * (victimShares + 1e6)` is exactly divisible by `(totalAssets + 1)`, avoiding rounding loss on their own mint.

Step 4 — Victim exits via burn().

The victim burns their shares. `burn()` computes proportional claims using raw `totalSupply` (no virtual offsets):

```
collateral = _totalCollateral.mulDiv(shares, _totalSupply);     // rounds down
debt = _totalDebt.mulDivUp(shares, _totalSupply);                // rounds up
```

The victim receives less collateral than they deposited, suffering loss both from the under-minting in Step 2 and the floor-division in `burn()`.

Step 5 — Attacker exits.

As the sole remaining shareholder, the attacker burns all shares and recovers **all** remaining collateral, because `VIRTUAL_SHARES` are ignored here. The attacker receives their deposit, the original donation, and the victim's rounding loss.

7.8.2 Concrete Example

Using a 6-decimal collateral token (USCC-like), 1:1 oracle, no debt:

Step 1 — Attacker donates `D = 1,000,000` via Morpho `supplyCollateral`.

State:

- `totalAssets = 1,000,000`
- `totalSupply = 0`

Step 2 — Victim deposits `V = 2`.

`sharesToMint = 2 * 1,000,000 / (1,000,000 + 1) = 2,000,000 / 1,000,001 = 1` (floor; fair value ≈ 1.999).

State:

- `totalAssets = 1,000,002`
- `totalSupply = 1`

Step 3 — Attacker deposits `A = 1,500,000`.

`sharesToMint = 1,500,000 * (1 + 1,000,000) / (1,000,002 + 1) = 1,500,001,500,000 / 1,000,003 = 1,499,997` (floor; negligible rounding loss).

State:

- `totalAssets = 2,500,002`
- `totalSupply = 1,499,998`

Step 4 — Victim burns their 1 share.

`collateral = 2,500,002 * 1 / 1,499,998 = 1` (floor; exact value ≈ 1.667). Victim deposited 2, receives 1. **Loss: 1 wei.**

State:

- `totalCollateral = 2,500,001`
- `totalSupply = 1,499,997`

Step 5 — Attacker burns all 1,499,997 remaining shares.

Receives all remaining collateral: `2,500,001`.

Total attacker cost: `1,000,000` (donation) + `1,500,000` (deposit) = `2,500,000`.

Net profit: 1 wei.

7.8.3 Conclusion

This is theoretically profitable as the `burn` operation allows the attacker to reclaim funds that should be held by the `VIRTUAL_SHARES`.

The `VIRTUAL_SHARES = 1e6` offset bounds the maximum extractable rounding loss per victim to approximately `totalAssets / 1e6`. To steal `X` from a victim, the attacker must donate at least `X * 1e6`. In the example above, the attacker locked 2,500,000 wei (~$2.50) to extract 1 wei(~$0.000001). This 2,500,000:1 cost-to-profit ratio makes the attack economically irrational at any meaningful scale.

Additionally, all of these flows can currently only be triggered through the `Facility` where they require cooperation from the Facilitator role.

7.8.4 Version 2 Changes

In Version 2, the virtual share offset was changed from a fixed `1e6` to a decimal-dependent value:

```
10 ** (18 - tokenDecimals)
```

This means 6-decimal tokens (e.g., USDC) now use `offset = 1e12`, while 18-decimal tokens (e.g., ETH) use `offset = 1`. The `VIRTUAL_ASSETS` constant remains `1`, and `burn()` still uses raw proportional math without virtual offsets: the core extraction mechanism is unchanged.

Furthermore, victim deposits can now round down to zero shares.

Our analysis across all three configurations shows the attack remains theoretically profitable in every case, but the economics vary dramatically by token decimals:

Scenario	Cost / Profit	Minimum Donation (0 victim shares)
Old code (offset = 1e6)	2,000,000	~2e24
New code, 6-dec USDC (offset = 1e12)	2,000,000,000,000	~2e22
New code, 18-dec ETH (offset = 1)	2	~2e19

18-decimal tokens are critically vulnerable. With `offset = 1`, the attacker needs only ~2x the victim's deposit to steal it entirely (cost/profit ratio of 2.0). For a victim depositing 10 ETH, the attacker donates ~20 ETH, the victim receives 0 shares, and the attacker recovers ~30 ETH on exit. This is 100,000x cheaper to attack than the old code.

6-decimal tokens are well-protected. With `offset = 1e12`, the cost/profit ratio is ~2 trillion, meaning the attacker must lock up 2 trillion times the profit. This is 1,000,000x more expensive to attack than the old code, making it economically infeasible at any scale.

The decimal-dependent offset effectively trades protection for one token class against another. The fundamental issue — `burn()` distributing collateral via raw `shares / totalSupply` without virtual offsets — remains the core extraction mechanism in all cases.

Code partially corrected:

In Version 4, the `burn()` function was updated to use the virtual offset when calculating collateral claims.

Hence, the security regarding profitable share inflation attacks is:

- 6-decimal debt tokens: virtual offset = 1e12, well-protected
- 18-decimal debt tokens: virtual offset = 1, better protected than in Version 2, but the attack can still be profitable if there is more than one victim depositor

As these functions are only used in the Facility, a profitable attack still requires (accidental) cooperation from the Facilitator role. The Facilitator can make sure that there can never be multiple victims and hence can prevent a profitable attack.

The new formula in `burn()` uses virtual shares, but it does not include virtual assets, which is technically inconsistent with the share computation in the minting logic.

The fact that 18-decimals debt tokens can be profitably attacked with two depositors and the fact that virtual assets are not part of the new `burn()` formula, is why this is marked as partially corrected.

7.9 USCCFund Is Missing Chainlink Price Freshness Check

Security | Low | Version 1 | Risk Accepted

CS-GRUNT-037

The `USCCFund.totalAssets()` function queries a Chainlink oracle via `latestRoundData()` to price the WUSCC supply and express total assets in USD terms. It validates that the answer is positive, the round is complete (`updatedAt != 0`), and the answer belongs to the current round (`answeredInRound >= roundId`).

However, the function does not enforce a time-based freshness constraint - there is no check that `block.timestamp - updatedAt` falls within an acceptable heartbeat or staleness threshold. The `answeredInRound >= roundId` check only confirms the round was answered, not that it was answered recently.

A stale price would cause `totalAssets()` to return an incorrect valuation. Currently, the result of `totalAssets()` is not used. However, future integration may rely upon it.

3F stated:

> The totalAssets value is not critical to our system's core operations.

Version 2:

The call to Chainlink was moved into an internal function `_getOraclePrice()`. In Version 2, this is further used to validate the order output amount in `_validateOutput()`. Stale oracle prices can lead to reverts if validation is too strict, or to loss of funds if orders get partially filled.

3F stated:

> That's why we have the possibility to increase the margin. It's not really the idea to have a very precise price.

7.10 USCCFund's totalAssets Reports Global wUSCC Supply Instead of Per-Fund Holdings

Correctness | Low | Version 1 | Acknowledged

CS-GRUNT-036

The `USCCFund.totalAssets()` function computes its return value using `WUSCC.totalSupply()`, which reflects the aggregate `wUSCC` minted across all `USCCFund` instances sharing the same `wUSCC` token. The contract documents this as intentional shared state, but the `IFund` interface and ERC-4626 convention both imply that `totalAssets()` returns the value managed by the specific instance being queried.

When multiple `USCCFund` instances exist, e.g., Fund A holding 1,000 `wUSCC` and Fund B holding 500 `wUSCC`, both report 1,500 `wUSCC` worth of assets. Each fund's `totalAssets()` is inflated by the other's deposits.

While the Facility's internal accounting is unaffected (it relies on balance snapshots rather than `totalAssets()`), any external consumer calling `totalAssets()` on a single fund will receive a value that double-counts assets held by sibling funds.

Acknowledged:

3F stated:

```
By nature, all the funds share the same totalAssets().
This is inherent to the wUSCC architecture.
```

7.11 Unbacked Yield Token Minting Leads to Yield Theft

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

CS-GRUNT-038

The `Request.authorizeMinting` function allows the owner or any account holding the consumer role to grant an address the right to mint a specified quantity of Principal Tokens (PT) and Yield Tokens (YT). However, the PT and YT amounts can be specified freely.

```
function mint() external {
    if (_syncWithdrawalStatus()) revert LibRequestErrors.AlreadyRepaid();
    (uint128 ptMintAuth, uint128 ytMintAuth) = msg.sender.mintAuth();
    msg.sender.updateMintAuth(0, 0);
    _asset().safeTransferFrom(msg.sender, address(this), ptMintAuth);
    _mint(msg.sender, ptMintAuth, ytMintAuth);
}
```

This allows a malicious or compromised consumer to first `authorizeMinting` for `0` PT and `2**128 - 1` YT tokens. The compromised account can then mint those YT tokens without making any payment *immediately before* the bridge loan is repaid. Next, they can claim nearly all the incoming yield.

Hence, a compromised consumer role can essentially steal all yield from prime brokers.

Code partially corrected / Risk accepted:

The function `Request.setRepaid()` was updated to enforce a cooling-off period after the last mint/consume, during which `setRepaid()` cannot be called. This prevents a compromised consumer from frontrunning the governance repayment call by minting unbacked YT tokens. However, the `_syncWithdrawalStatus()` function still sets repaid to true when `block.timestamp >= repaymentDeadline` without enforcing this delay, so a compromised consumer can mint unbacked YT tokens shortly before the repayment deadline.

Additionally, in case a consumer compromise is detected by the governance, the governance has no recourse besides withholding all yield. This affects all YT holders.

The code fix therefore needs to be accompanied by off-chain monitoring and a repayment deadline set sufficiently far in the future.

8 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	1

- Stale Morpho Market Data Code Corrected

Medium-Severity Findings	16

- Facilitator Overpays Request to Drain LP Deposits Code Corrected
- Partial Pre-Liquidations Revert Due to Rounding Mismatch With Morpho Code Corrected
- Permanent Blocking of Setting Status to Repaid Code Corrected
- Burn Divides by Total Debt Code Corrected
- Burn Reverts Due to Rounding Errors Code Corrected
- Facilitator Can Set Return Amount to 0 Code Corrected
- Facilitator Steals From Other Intents via withdrawManager Code Corrected
- Facilitator Steals From Prime Brokers Code Corrected
- Facilitator Theft With Malicious Request Code Corrected
- Malicious Facilitator Can Drain Intent via setFund Code Corrected
- Missing Cooldown on Rebalance Code Corrected
- NAV Calculation Includes Bad Debt Positions Code Corrected
- Operator Triggers Permanent State Deadlock Code Corrected
- Partial Liquidations Fail Health Check Code Corrected
- Repaid Shares Can Exceed Borrow Shares Code Corrected
- YT Approval Resets PT Allowance Code Corrected

Low-Severity Findings	37

- Malicious Consumer Bypasses mintToRepaidDelay via Governance Reentrancy Code Corrected
- Burn Can Increase PM LTV Code Corrected
- Borrowing After Supplying collateralForBorrow Can Fail the Health Check Code Corrected
- Proportional Withdrawal Ignores Target LTV Code Corrected
- Repaying totalBorrowed() Can Revert Due to Rounding Mismatch Code Corrected
- Reverting a Deposit Might Be Ineffective Risk Mitigated
- Withdrawing availableCollateral Can Fail the Health Check Code Corrected
- maxBorrow Returns an Unborrowable Amount Code Corrected
- Burning All Shares Reverts With Depleted Principal Assets Code Corrected
- Code Duplication and Inconsistencies in Oracle Initialization Code Corrected
- Curator Can Insert Duplicate Entries in Supply and Withdrawal Queues Code Corrected

- Facility Transfers Zero-Amount Tokens When Share Proportion Rounds Down Code Corrected
- Front-Running Prime Broker Deposits Code Corrected
- Incomplete ERC-4626 Compliance for ControlledVault Code Corrected
- Incorrect Event Emission During Rebalancing Code Corrected
- Incorrect NatSpec Comment on resolveStart Code Corrected
- Maximum Management Fee Cap of 50% Is Excessively Permissive Code Corrected
- Missing State Checks for setFund and setRequest Specification Changed
- Morpho Reference Can Be Immutable Code Corrected
- No PositionManager LLTV Check Enforced During Deposits Code Corrected
- Off-by-one in Offer Expiration Check Code Corrected
- One Fund Simultaneously Assigned to Multiple Intents Code Corrected
- Paused Position Manager Allows Deposits and Withdrawals When totalAssets Is Unchanged Code Corrected
- Performance Fees Charged on Gross Gain Instead of Net Gain Code Corrected
- PositionManager Deposits and Withdrawals Revert on Small Asset Deltas Code Corrected
- Prime Broker Can Front-Run authorizeMinting to Double-Mint Risk Mitigated
- Redundant Deletions in USCCFund Code Corrected
- Redundant Storage in USCCFund Code Corrected
- Role Constants Are Public in USCCFund and TransferGuard Code Corrected
- Target PM Restrictions Unchecked When Deposit PM Is Guard Key Risk Mitigated
- TokenController Reverts on Self-Transfers Deviating From ERC-20 Standard Code Corrected
- USCCFund Sends Assets to Msg.Sender Instead of Order.Receiver Code Corrected
- WrappedAsset Initialization Allows Decimal Mismatch Code Corrected
- _settleShares Rounds Down Shares Burned Code Corrected
- addBorrowModule and removeBorrowModule Skip Fee Accounting Code Corrected
- repaymentDeadline Not Validated Code Corrected
- setRequest Can Bind the Same Request to Multiple Intents Code Corrected

Informational Findings	27

- Incorrect Comment About Fee Shares Code Corrected
- Misnamed Custom Error Code Corrected
- Pause Duration Is One Second Longer Than Specified Code Corrected
- Redundant Calls to Accrue Interest Code Corrected
- Unchecked Return Value in removeBorrowModule Code Corrected
- Unreachable ZeroShares Revert in _settleShares Code Corrected
- Arbitrary Borrower Address in preLiquidate Code Corrected
- Arbitrary Decimals in PositionManager Code Corrected
- Inconsistent Validation for Order's Input Amount Code Corrected
- Incorrect Error Referenced Code Corrected
- Incorrect Return Values in USCCFund Code Corrected

- Incorrect Safety Comment in TokenController Code Corrected
- Initialization Not Disabled in Implementation Contracts Code Corrected
- Misleading Error State in USCCFund Revert Messages Code Corrected
- Missing Asset Compatibility Check in MorphoBorrowPosition Initialize Code Corrected
- Missing Function to Query Accurate Share Price Code Corrected
- Missing Sanity Checks on LLTV Code Corrected
- Missing Verification of Target Order Code Corrected
- Missing Zero-Address Check in Claim and Withdraw Code Corrected
- No Upper Bound on setMaxRebalanceLoss Code Corrected
- Outdated NatSpec Documentation Code Corrected
- Solidity Version Not Fixed Code Corrected
- Typo in _intialPmParameters Variable Code Corrected
- USCCFund Uses Potentially Stale Superstate Allowlist Interface Code Corrected
- Unnecessary Free Memory Pointer Update Code Corrected
- Unnecessary cachedBalance in USCCFund Logic Code Corrected
- addBorrowModule Lacks Validation Code Corrected

8.1 Stale Morpho Market Data

Correctness High Version 1 Code Corrected

CS-GRUNT-001

The contract `MorphoBorrowPosition` reads market state from Morpho via `IMorpho.market()` and `IMorpho.position()`, which return cached storage values that do not include interest accrued since the last state-changing interaction. Morpho's own NatSpec explicitly warns:

"Warning: totalBorrowAssets does not contain the accrued interest since the last interest accrual."

The stale `totalBorrowAssets` propagates through the share-to-asset conversion (`toAssetsUp`) into every function that computes borrowed amounts, causing debt to be systematically understated. This produces a cascading effect: health checks are biased toward reporting positions as healthy, borrowing capacity is overstated, and available collateral is overstated.

Most critically, `preLiquidate()` performs its health check before calling `accrueInterest()`, meaning the guard that determines whether a position is liquidatable uses stale data. This can block legitimate pre-liquidations, defeating the purpose of the dual-LTV early liquidation mechanism.

A liquidator could call `morpho.accrueInterest(marketParams)` in the same transaction before `preLiquidate()`, but this is an undocumented requirement that standard liquidation bots might not know to perform.

The staleness window equals `block.timestamp - market.lastUpdate`, which grows with market inactivity. In illiquid markets this gap can be days or weeks, making `totalBorrowAssets` significantly understated.

The incorrect data propagates further through the contract:

- `totalAssets()` are overestimated.
- As a consequence, fee amounts are incorrectly computed.
- `_settleShares()` uses stale values.

- As a consequence, minted/burnt share amounts are incorrectly computed.

As a `PositionManager` will be used by multiple intents, one intent might profit at the cost of another.

Consider the following example with a single `PositionManager`, two intents and a single `MorphoBorrowPosition`.

1. The two intents each have half the shares.
2. The `MorphoBorrowPosition` is very big and very stale. 50,000 USDC of unaccounted interest has accumulated.
3. The facilitator decides to reduce the debt of intent 1 and calls `withdrawManager` with `repayAmount` = 50,000
4. `PositionManager.withdraw` is called:
    1. `_accrueFees` overestimates the `totalAssets()` by 50,000 USDC and hence pays out too much management fee and performance fee
    2. `_accrueFees` measures `totalAssetsBefore` which is 50,000 USDC too high
    3. `processWithdrawal` reduces the debt by 50,000 USDC
        1. It calls `repay` on Morpho which will accrue the interest
    4. At the end `_settleShares` is called and observes a `totalAssetsBefore == totalAssetsAfter` because the reduced debt was offset by the accrued interest
    5. Intent 1 gets no shares, and hence the share price is increased. Intent 2 basically receives 25,000 USDC from Intent 1.

Lastly, because the health checks use stale data, functions like `MorphoBorrowPosition.withdraw` can create positions that are immediately (pre-)liquidatable. This could happen as part of rebalancing and leads to a direct loss of funds.

Code corrected:

State-changing functions in `MorphoBorrowPosition` now call `morpho.accrueInterest()` before reading market and position data. View functions use `MorphoBalancesLib` to compute expected balances with accrued interest.

8.2 Facilitator Overpays Request to Drain LP Deposits

Trust | Medium | Version 2 | Code Corrected

CS-GRUNT-081

A compromised Facilitator can front-run the Request owner's `setRepaid()` call by calling `FacilityRequests.repay()` to push the intent's entire asset balance into the Request.

This is most critical in a redemption intent (PM shares to asset), where position manager shares are exchanged for the underlying asset, leaving the intent holding a large asset balance. The Facilitator can repay this full balance into the Request just before `setRepaid()` is called.

Once the Request is marked as repaid, YT holders redeem against the inflated balance, thereby receiving assets that should have been claimable by the intent's depositors. When LPs call `claim()`, the

intent holds no assets and they receive nothing. The Facilitator could acquire YT tokens to extract some of the value stolen.

Code corrected:

`Request.setRepaid` now accepts both a `minBalance` and a `maxBalance` parameter and reverts if the contract's asset balance exceeds `maxBalance`, preventing a compromised Facilitator from inflating the redemption value by over-repaying into the `Request`. Note that only the `Request` owner can call `setRepaid` which is fully trusted. Hence, those values are assumed to be set correctly and to protect intent share holders.

Code comments suggest that the caller could set `maxBalance` to `type(uint256).max` to skip the upper bound check. It must be noted that doing so would re-enable the aforementioned attack vectors, requiring a trust model in which the Facilitator is fully trusted.

8.3 Partial Pre-Liquidations Revert Due to Rounding Mismatch With Morpho

Correctness | Medium | Version 2 | Code Corrected

CS-GRUNT-085

`MorphoBorrowPosition.preLiquidate` allows liquidators to partially repay an unhealthy position's debt and seize a proportional amount of collateral. After the debt is repaid, the contract withdraws collateral from Morpho, which triggers Morpho's internal `_isHealthy` check on the remaining position.

The function `_computeSeizedAssets` is supposed to compute the maximum collateral that may be seized while passing the Morpho health check. `_computeSeizedAssets` works by combining the oracle price and the market LLTV into a single product with one floor: `floor(price * lltv / WAD)`. Morpho's `_isHealthy`, however, applies two sequential floors:

- first `floor(collateral * price / ORACLE_PRICE_SCALE)`,
- then `floor(result * lltv / WAD)`.

This can cause Morpho's valuation to fall 1 unit below the value assumed by `_computeSeizedAssets`.

As a result, partial pre-liquidations can revert when Morpho's stricter rounding deems the post-seizure position unhealthy, even though `_computeSeizedAssets` attempted to compute a valid `seizedAssets` value. This blocks meaningful partial liquidations for positions above the pre-liquidation LTV.

Code corrected:

The computation of `seizedAssets` in `_computeSeizedAssets` has been updated to counteract the sequential flooring of Morpho's health check. This ensures that the seized collateral amount will not lead to a revert.

8.4 Permanent Blocking of Setting Status to Repaid

Security | Medium | Version 2 | Code Corrected

CS-GRUNT-082

`Request.setRepaid()` can only be called after `mintToRepaidDelay` seconds have elapsed since the last mint:

```
uint40 _availableAt = _lastMint + req.mintToRepaidDelay;
if (block.timestamp < _availableAt) {
    revert LibRequestErrors.MintToRepaidDelayNotElapsed(_availableAt);
}
```

`Request.mint()` updates `lastMintTimestamp` to the current `block.timestamp` on every call. Because the function does not revert on zero amounts, any address can call `mint(0, 0)` to reset the delay. An attacker can repeat this before each delay window expires, permanently preventing the owner from calling `setRepaid`. Minting zero tokens does not require any minting authorization, so the owner cannot prevent this from happening.

Therefore, a request cannot be marked as repaid and hence:

- PT/YT holders cannot redeem
- Intent shareholders cannot claim

This is only resolved once the `repaymentDeadline` is reached.

Code corrected:

In Version 4 `Request.mint` returns early when the caller has no minting authorization (both `ptMintAuth` and `ytMintAuth` are zero).

8.5 Burn Divides by Total Debt

Correctness | Medium | Version 1 | Code Corrected

CS-GRUNT-002

When burning shares, the `PositionManager` proportionally repays debt and withdraws collateral from all modules in the withdrawal queue.

The function `LibOperations.processBurn()` iterates over all modules in the withdrawal queue and repays the proportional debt and claims the corresponding collateral. For each module, the proportional debt of a module is calculated as:

`toRepay = debtToRepay * positionDebt / totalDebt`

`LibView.debtAmount()` is used to calculate the `totalDebt` in that formula. However, `debtAmount()` takes the debt of all registered borrow modules and **not just the modules in the withdrawal queue**.

```
/// @return amount The total debt amount
function debtAmount(PositionManagerStorageData storage ps) internal view returns (uint256 amount) {
  address[] memory modules = ps.borrowModules.values();
  uint256 modulesLength = modules.length;
  for (uint256 i = 0; i < modulesLength;) {
    amount += IBorrowPosition(modules[i]).totalBorrowed();
    unchecked {
      ++i;
    }
  }
}
```

Therefore, if not all modules are in the withdrawal queue, inside `processBurn`:

- the sum of the `positionDebt` values will not equal `totalDebt`

- hence, the sum of the `toRepay` values will not equal `debtToRepay`
- hence, not all the intended debt will be repaid
- instead, debt tokens are stuck in the `PositionManager` contract

Luckily, in many cases instead of stuck funds, the result will be an incorrect revert. This is because for the same reason that not enough debt is repaid, not enough collateral will be collected. Therefore, the collateral transfer in the `burn` function will fail.

Theoretically, an attacker could supply the missing collateral to grief the burning user in this case.

Code corrected:

The function `processBurn()` was removed in Version 2. The `_withdrawProportional` function, which has a similar functionality, computes withdrawal amounts using the cumulative debt of the withdrawal queue modules rather than the total debt of all borrow modules.

```
address[] memory queue = _storage.withdrawalQueue;
uint256 queueLength = queue.length;
...
for (uint256 i = 0; i < queueLength; i++) {
  cumDebts[i] = (i > 0 ? cumDebts[i - 1] : 0) + IBorrowPosition(queue[i]).totalBorrowed();
  ...
}
```

8.6 Burn Reverts Due to Rounding Errors

Correctness | Medium | Version 1 | Code Corrected

CS-GRUNT-003

When the Facilitator exits a position via `PositionManagerLP.burnManager`, the internal `LibOperations.processBurn` function is invoked to handle the proportional withdrawal of collateral from underlying borrow positions. This function iterates through each position in the `withdrawalQueue`, calculating and withdrawing a share of the total collateral.

Since the division is performed on a per-position basis across potentially several modules, and perfect divisibility is not guaranteed, even a single instance of a non-perfect division will result in a recovery shortfall.

```
// Withdraw proportionally
if (remainingCollateral > 0 && positionCollateral > 0 && totalCollateral > 0) {
  uint256 toWithdraw = collateralToWithdraw.mulDiv(positionCollateral, totalCollateral);
  if (toWithdraw > 0) {
    position.withdraw(toWithdraw);
    remainingCollateral -= toWithdraw;
  }
}
```

`PositionManagerLP.burnManager` optimistically assumes the withdrawal will be exact. It calculates the total `collateralToWithdraw` upfront but never verifies the actual amount of collateral recovered after the `LibOperations.processBurn` call. Consequently, the final amount of collateral held by the contract is almost always less than the `collateralToWithdraw` value, and the subsequent attempt to transfer the full `collateralToWithdraw` to the Facilitator reverts due to insufficient funds.

Code corrected:

The function `processBurn()` was removed in Version 2. The `_withdrawProportional` function, which has a similar functionality, avoids such rounding errors and obtains exactly the requested amount of collateral.

8.7 Facilitator Can Set Return Amount to 0

Trust | Medium | Version 1 | Code Corrected

CS-GRUNT-004

The `USCCFund` contract is responsible for managing the state transitions of orders and wrapping received tokens (USCC) into a centrally controlled wrapped asset (wUSCC).

Within the `USCCFund._state()` function, for a deposit order to transition from the `PROCESSING` state to the `UNLOCKING` state, the contract verifies if the amount of USCC received is at least the expected output `_effectiveOutput`.

Critically, an order can be created with an `order.output = 0` via the `FacilityFunds.create` function. By setting an order's expected output to zero, the state transition check in `_state()` is bypassed. This allows an order to be prematurely considered complete and eligible for unlocking as soon as it is committed. A malicious or compromised `Facilitator` can exploit this vulnerability to steal user funds through the following steps:

1. **Malicious Order Creation**: The `Facilitator` calls `FacilityFunds.create` on the `USCCFund`, providing valid `order.input` (e.g., 1,000,000 USDC representing user deposits and prime broker bridge loans) but intentionally sets `order.output` to 0.
2. **Fund Commitment**: The `Facilitator` then calls `FacilityFunds.commit`, transferring the 1,000,000 USDC to the Superstate. The order's state becomes `PROCESSING`.
3. **Bypass State-Transition Check**: Due to the zero `_effectiveOutput`, the order immediately becomes eligible for unlocking. The `_state()` function returns `UNLOCKING`, even though no actual USCC has been received for this specific order.
4. **Order Finalization**: The `Facilitator` calls `FacilityFunds.unlock` to finalize and close this specific order. At this point, no wUSCC is minted.
5. **Establish Malicious Intent**: The `Facilitator` creates a *separate* intent, which only they deposit into. This intent targets the same `USCCFund`. This new intent is designed to receive wUSCC from the `USCCFund` based on the previously transferred 1,000,000 USDC.
6. **Execute the Theft of wUSCC**: Once the Superstate processes the initial deposit and transfers the corresponding USCC back to the `USCCFund`, the `Facilitator` can call `FacilityFunds.unlock` for the *second* intent. That mints wUSCC to the second intent which is entirely owned by the `Facilitator`. The USCC that was originally meant for the first order. Therefore, the `Facilitator` effectively steals the user's funds.

Code corrected:

`USCCFund.create()` now calls `_validateOutput()` which checks the order output against an oracle price, reverting if the deviation exceeds `MAX_OUTPUT_DEVIATION` (500 bps). This prevents a facilitator from creating orders with artificially low output amounts.

8.8 Facilitator Steals From Other Intents via withdrawManager

Trust Medium Version 1 Code Corrected

CS-GRUNT-062

The Facility holds pooled assets across multiple intents, and any facilitator can create new intents with arbitrary PositionManager contracts. There is no validation that a PositionManager's reported balances reflect genuine external positions rather than temporarily injected funds.

A malicious facilitator exploits this by inflating their intent's credited balance with flash-loaned funds, then resolving the intent to claim assets belonging to other intents. The attack proceeds as follows:

1. The facilitator creates a new intent with a malicious PositionManager they control, setting both `collateralAsset` and `debtAsset` to the same token (e.g. USDC). The facilitator is the only depositor in this intent.
2. The facilitator flash-loans a large amount of USDC (e.g. 1M) directly into the malicious PositionManager.
3. The facilitator calls `withdrawManager` through the Facility. The Facility does snapshot-based accounting, but because `collateralAsset == debtAsset` it will double-count the 1M USDC the facilitator withdraws. Therefore, the Facility receives 1M USDC, but credits the facilitator's intent with 2M USDC.
4. The facilitator resolves the intent and claims the inflated balance (e.g. 2M USDC, the 1M flash-loaned plus 1M from other intents).
5. The facilitator repays the flash loan and keeps the difference stolen from other intents.

Code corrected:

`FacilityPositionManager._initialPmParameters()` validates that `collateralAsset`, `debtAsset`, and the `positionManager` address are all distinct. Thereby, it prevents the double-counting attack.

8.9 Facilitator Steals From Prime Brokers

Trust Medium Version 1 Code Corrected

CS-GRUNT-005

Although the Facilitator is trusted to choose correct parameters, the trust model assumes it should not be able to unilaterally steal assets.

The Facilitator can call `pull()` in `FacilityRequests`, which invokes `pullFunds()` on the attached `Request`. `pullFunds()` is only callable while the Request has not yet been marked as repaid.

A malicious (or compromised) Facilitator can front-run the Request owner's pending `setRepaid()` transaction by calling `pull()` with the full asset balance of the Request. Afterwards, `setRepaid()` gets executed and unlocks PT/YT redemptions, but the Request now holds no assets. Consequently, PT and YT holders (the Prime Brokers) receive nothing when they attempt to redeem.

Therefore, the Facilitator can essentially steal all funds from Prime Brokers.

Code corrected:

`Request.setRepaid()` now accepts a `minBalance` parameter and reverts if the contract's asset balance is below the specified threshold. If a compromised Facilitator front-runs `setRepaid()` by draining the Request via `pull()`, the reduced balance causes the call to `setRepaid()` to revert, giving the Request owner time to intervene and revoke the Facilitator.

8.10 Facilitator Theft With Malicious Request

Trust | Medium | Version 1 | Code Corrected

CS-GRUNT-006

Although the Facilitator is trusted to choose correct parameters, the trust model assumes they should not be able to unilaterally steal all user assets. However, because `repay()` in `FacilityRequests` accepts an arbitrary `amount` and `setRequest()` allows attaching any contract whose asset matches the PositionManager's debt asset, a malicious or compromised Facilitator can drain all deposited assets from an intent within two blocks.

In the first block, the Facilitator:

1. Calls `lock()` on the target intent, advancing it from **Depositing** to **Resolving**.
2. Deploys a new `Request` via `RequestFactory` with a `repaymentDeadline` of `block.timestamp + 12` and themselves as owner.
3. Calls `authorizeMinting()` on the new Request, then `mint()` to issue PT and YT tokens to themselves.

In the next block, after `repaymentDeadline` has passed:

1. Calls `setRequest()` to attach the malicious Request to the intent.
2. Calls `repay()` with the intent's full asset balance, transferring all deposited funds to the attacker-controlled Request.
3. Redeems all PT and YT tokens via the Request's ERC-4626 vaults, receiving the stolen assets. Redemption succeeds because `repaymentDeadline` has already elapsed.

As a result, all LP deposits held by the targeted intent are stolen.

Code corrected:

`setRequest()` in `FacilityIntents` now requires a quorum of guardian signatures before a request can be attached to an intent, preventing the facilitator from unilaterally setting a malicious request.

8.11 Malicious Facilitator Can Drain Intent via setFund

Trust | Medium | Version 1 | Code Corrected

CS-GRUNT-007

The `setFund` function on an intent is callable by the facilitator role and accepts any contract that satisfies the asset-matching checks. There is no timelock or additional validation on the fund contract beyond interface conformance, allowing the facilitator to point an intent at an arbitrary contract they control.

A malicious facilitator can call `setFund` with a fund contract that forwards all received assets to the attacker. They can then create an order and commit it in the same block, draining the intent's entire balance before depositors or more privileged roles can react.

This gives the facilitator role unilateral power to steal all funds held by any intent. Since the attack can be executed atomically within a single block, there is no window for governance intervention or user withdrawal.

Code corrected:

`setFund()` in `FacilityIntents` now requires a quorum of guardian signatures before a fund can be attached to an intent, preventing the facilitator from unilaterally setting a malicious fund contract.

8.12 Missing Cooldown on Rebalance

Trust | Medium | Version 1 | Code Corrected

CS-GRUNT-008

A special role called rebalancer can rebalance the debt positions across the multiple Morpho positions. The `PositionManager` enforces that any loss generated from the rebalancing does not exceed a certain threshold:

```
if (totalAssetsAfter < totalAssetsBefore) {
  uint256 loss = totalAssetsBefore - totalAssetsAfter;
  // loss * BPS / totalAssetsBefore > maxRebalanceLoss
  // Rearranged to avoid division: loss * BPS > maxRebalanceLoss * totalAssetsBefore
  if (loss * BPS > uint256(_storage.maxRebalanceLoss) * totalAssetsBefore) {
    revert LibManagerErrors.RebalanceLossExceedsMax();
  }
}
```

However, the rebalance operation has no cooldown. A malicious or compromised rebalancer can repeatedly rebalance to extract more than the loss threshold. For example, if the maximum rebalance loss is set to 1%, calling the function 200 times would extract approximately $100\% - 99\%^{200} \approx 87\%$ of the total value. This means the `maxRebalanceLoss` is an ineffective protection for a semi-trusted role.

Code corrected:

`PositionManagerRebalancing.rebalance()` enforces a configurable cooldown period between consecutive rebalancing operations, preventing repeated calls:

```
uint40 cooldown = _storage.rebalanceConfig.rebalanceCooldown;
uint40 lastRebalance = _storage.rebalanceConfig.lastRebalanceTimestamp;
if (cooldown > 0 && lastRebalance > 0) {
  ...
  if (uint40(block.timestamp) - lastRebalance < cooldown) {
    revert LibManagerErrors.RebalanceCooldownNotElapsed();
  }
}
```

The cooldown duration is configurable via `PositionManagerAdmin.setRebalanceConfig()`. The cooldown can be set to zero to disable this protection entirely. This makes the rebalancer fully trusted.

8.13 NAV Calculation Includes Bad Debt Positions

Design | Medium | Version 1 | Code Corrected

The total net asset value (NAV) of the `PositionManager` is calculated by aggregating the collateral value quoted in debt tokens and the total debt, and then netting them against each other:

```
function totalAssets(PositionManagerStorageData storage ps) internal view returns (uint256) {
  return collateralAmountQuoted(ps).zeroFloorSub(debtAmount(ps));
}
```

If a position's debt exceeds its collateral (i.e. "bad debt"), its negative value reduces the total NAV. However, 3F states that bad debt positions should be abandoned instead of repaid. Abandoning a position is equivalent to treating it as having zero value, yet the current implementation accounts for it as negative. As a result, the total NAV is incorrectly understated by the bad debt amount.

Code corrected:

`totalAssets()` in `LibView` now iterates over each borrow module individually and floors each module's net value at zero rather than netting collateral and debt globally:

```
function totalAssets(PositionManagerStorageData storage ps) internal view returns (uint256 amount) {
  address[] memory modules = ps.borrowModules.values();
  uint256 modulesLength = modules.length;
  for (uint256 i = 0; i < modulesLength; ++i) {
    uint256 collateral = IBorrowPosition(modules[i]).totalCollateralQuoted();
    uint256 debt = IBorrowPosition(modules[i]).totalBorrowed();
    amount += collateral.zeroFloorSub(debt);
  }
}
```

This ensures that bad-debt positions (where debt exceeds collateral) contribute zero rather than a negative amount to the total NAV.

8.14 Operator Triggers Permanent State Deadlock

Correctness **Medium** **Version 1** **Code Corrected**

The `USCCFund` contract manages asynchronous deposit and redemption orders through a state machine. An operator can call `USCCFund.recover()` to set an order's internal state to `State.RECOVERING`, signaling that the off-chain operation has failed and that sent funds are getting returned by Superstate.

However, if the order ends up being successfully executed by Superstate eventually, the state machine enters a deadlock. The `_state` function returns `State.PROCESSING`:

```
if (_internalState == State.RECOVERING) {
  uint256 _amount;
  if (order.mode == Mode.DEPOSIT) {
    // Deposit: check if we can recover USDC
    _amount = USDC.balanceOf(address(this));
    return _amount >= _effectiveInput ? (State.RECOVERING, _amount) : (State.PROCESSING, 0);
  } else {
    // Redeem: check if we can recover USCC
    _amount = USCC.balanceOf(address(this)).zeroFloorSub(_storage.cachedBalance);
    return _amount >= _effectiveInput ? (State.RECOVERING, _amount) : (State.PROCESSING, 0);
  }
}
```

However, the two functions available to transition the state (`recover` and `unlock`) cannot handle this scenario. `recover()` reverts unless the state is `RECOVERING`, and `unlock()` reverts unless the state is

UNLOCKING. With both state transition paths blocked, the order stays permanently stuck in state RECOVERING.

Code corrected:

The USCCFund contract now includes a cancelRecovering() function callable by OPERATOR_ROLE or owner when the state is RECOVERING to manually reset the state to PROCESSING. This later allows the fund to transition to UNLOCKING.

8.15 Partial Liquidations Fail Health Check

Correctness Medium Version 1 Code Corrected

CS-GRUNT-011

Pre-liquidations are possible when the health of a position (debt / (collateral * price)) crosses the specified pre-liquidation threshold. If the health then becomes greater than the LTV threshold of the Morpho market, native Morpho liquidations also become possible. For positions that are above the pre-liquidation and the Morpho market LLTV, the following issue exists:

Pre-liquidations first repay debt of a Morpho position and then withdraw the corresponding collateral. Withdrawing collateral triggers a health check within Morpho.

However, pro-rata pre-liquidations keep the LTV (debt / (collateral * price)) of the position unchanged: repaying a fraction of the debt and removing the same fraction of collateral results in the same health. As partial pre-liquidations do not improve the health, the reduced position still fails Morpho's health check when collateral is withdrawn, causing any partial pre-liquidation to revert.

Therefore, partial pre-liquidations are not possible when the position health is above the Morpho market LLTV.

Code corrected:

preLiquidate() now either caps the seized collateral (via _computeSeizedAssets()) or extends the repaid shares (via _computeRepaidShares()) to ensure that a partially liquidated position still passes Morpho's health check.

```
// Use the greater of proportional and minimum required, capped at total borrow shares
repaidShares = proportional.max(minShares).min(borrowShares);

...

// Use the lesser of proportional and maximum allowed
seizedAssets = proportional.min(maxSeized);
```

8.16 Repaid Shares Can Exceed Borrow Shares

Correctness Medium Version 1 Code Corrected

CS-GRUNT-012

MorphoBorrowPosition.preLiquidate() allows any caller to liquidate an unhealthy position. When invoked with a non-zero seizedAssets amount, the corresponding number of shares to repay is rounded up:

```
repaidShares = seizedAssets.mulDivUp(position.borrowShares, position.collateral);
```

This can cause `repaidShares` to exceed `position.borrowShares` (i.e., the total outstanding borrow shares of the position). When the inflated `repaidShares` value is passed to Morpho's `repay` function, Morpho subtracts it from the stored share balance. The resulting subtraction can underflow and the transaction can revert.

Hence, a call to `preLiquidate` can fail if a `seizedAssets` value equal to the position's collateral assets is provided. This is unexpected for integrators and should be documented.

From a liquidator's perspective, this can also happen on a partial liquidation when there has been a previous partial liquidation leaving behind a position with exactly the right amount of `seizedAssets`.

Code corrected:

The `preLiquidate()` function in `MorphoBorrowPosition` caps the repaid shares by the position's borrow shares.

8.17 YT Approval Resets PT Allowance

Correctness | Medium | Version 1 | Code Corrected

CS-GRUNT-013

When a user wants to grant an approval on the YT token, `ControlledToken.approve()` is called, which delegates to the public `_approve` function in `TokenController`.

```
function _approve(address from, address spender, uint256 amount, bool yt)
            public virtual returns (bool) {
    _checkToken(yt);
    uint256 ptAmount = yt.ternary(0, amount);
    uint256 ytAmount = yt.ternary(amount, 0);
    return _setAllowance(from, spender, ptAmount, ytAmount);
}
```

When the YT token invokes `_approve`, the `yt` flag is `true`, causing `yt.ternary(0, amount)` to evaluate `ptAmount` to zero. Both `ptAmount` and `ytAmount` are then forwarded to `_setAllowance`, which unconditionally overwrites the spender's stored PT allowance with the supplied `ptAmount` value of zero, regardless of any previously set PT allowance.

As a result, any existing PT allowance granted to a spender is silently reset to zero whenever the token owner grants a YT allowance to that same spender, potentially breaking integrations. Similarly, approving the PT token resets the YT allowance by the same mechanism. Only `TokenController.approveBatch()` can successfully set allowances for both tokens simultaneously.

Code corrected:

The `_approve()` function in `TokenController.sol` reads the existing allowances via `from.allowances(spender)` and preserves the sibling token's allowance in `_setAllowance()`.

8.18 Malicious Consumer Bypasses mintToRepaidDelay via Governance Reentrancy

Design | Low | Version 5 | Code Corrected

CS-GRUNT-105

The fully-trusted `owner` of a `Request` is expected to be a multi-signature wallet. Most multi-signature wallets have unpermissioned execution, meaning that once quorum is reached, anyone can execute the transaction on-chain.

A malicious or compromised consumer can call `consume()` and, inside the `onRequestConsumed` callback, execute a pending `setRepaid()` multi-signature call. Since `lastMintTimestamp` is only updated *after* the callback, the `mintToRepaidDelay` check in `setRepaid()` passes against the stale timestamp and marks the request as repaid.

Crucially, this does not merely bypass the delay. It breaks an invariant of the `Request` contract. After `setRepaid()` executes inside the callback, the `repaid` flag is set to `true` and PT/YT holder withdrawals are enabled. However, when the callback returns, `consume()` continues execution: it pulls assets from the maker and mints new PT and YT tokens. This means PT and YT tokens are minted *after* the request has already been marked as repaid.

This violates the core assumption that no new PT/YT can be created once repayment is finalized.

Code corrected:

A `nonReentrant` modifier was added to `setRepaid()`. Combined with the existing `nonReentrant` modifier on `consume`, this prevents an attacker from re-entering `setRepaid()`.

8.19 Burn Can Increase PM LTV

Design | Low | Version 3 | Code Corrected

CS-GRUNT-083

The function `PositionManagerLp.burn()` calls `LibOperations.processWithdrawal()` with `checkLtv` set to false, assuming that a proportional debt/collateral withdrawal in `_withdrawProportional` leaves the per-position LTV of touched positions unchanged.

However, this only holds when the withdrawal queue is full, i.e., all borrow modules are included in the queue. If the withdrawal queue contains less collateralized modules compared to those outside of it, the proportional claim would disproportionately reduce collateral relative to debt, thereby increasing the LTV of the remaining position.

Overall, this means that a `burn` operation could re-leverage the positions of the `PositionManager` which is unintended.

Code corrected:

`PositionManagerLP.burn` now conditionally sets `skipLtvCheck` based on whether the withdrawal queue covers all borrow modules, only skipping the LTV check when the queue is full.

8.20 Borrowing After Supplying collateralForBorrow Can Fail the Health Check

Correctness | Low | Version 2 | Code Corrected

CS-GRUNT-098

`MorphoBorrowPosition.collateralForBorrow(toBorrow, ltv)` computes the collateral needed so that supplying it and then borrowing `toBorrow` keeps the position at or below the target LTV. However, calling `supply(collateralForBorrow(toBorrow, ltv))` followed by `borrow(toBorrow)` can revert because the post-borrow `_isHealthy` check may deem the position unhealthy.

Two independent rounding mismatches contribute to this failure. First, `collateralForBorrow` predicts the post-borrow debt as `toAssetsUp(oldBorrowShares, tba, tbs) + toBorrow`, but Morpho's `borrow()` converts `toBorrow` to shares via `toSharesUp` (rounding up), and the subsequent `_isHealthy` check reconverts the new total shares back to assets via `toAssetsUp`. This share round-trip can inflate the actual debt beyond the predicted value.

A fresh position borrowing 2 assets can end up with an actual debt of 3 after the double conversion. Second, even when the share round-trip is exact, `_requiredCollateral` inverts a single combined division while `_isHealthy` applies two separate floor operations, the same double-floor mismatch class as the `availableCollateral` and `preLiquidate` issues.

Both sources can compound, producing a shortfall greater than 1 unit. This means the collateral amount returned by `collateralForBorrow` is insufficient to pass the health check after the borrow, causing the transaction to revert. This can happen as part of:

- rebalancing operations
- deposit operations into the `PositionManager`, in `processDeposit` if `PositionManager.ltv == safeLTV`

Code corrected:

Two changes have been made to address this issue:

1. `collateralForBorrow` now uses `toSharesUp(toBorrow, tba, tbs)` to predict the post-borrow shares and out of those, the post-borrow assets. This ensures that the share round-trip in the borrow and health check is consistent.
2. `_requiredCollateral` has been updated to apply the same double-floor logic as `_isHealthy` when inverting the price and LLTV, ensuring that the collateral requirement is computed with the same rounding assumptions as the health check.

Together these changes resolve the finding.

8.21 Proportional Withdrawal Ignores Target LTV

Correctness | Low | Version 2 | Code Corrected

CS-GRUNT-096

`PositionManager` supports two withdrawal strategies: sequential and proportional. The sequential strategy (`_withdrawSequential`) correctly caps each position's collateral withdrawal using `availableCollateral(_storage.ltv)`, which accounts for the manager's target LTV and ensures positions remain within their intended leverage after withdrawal.

The proportional strategy (`_withdrawProportional`) distributes collateral withdrawals across positions proportional to each position's `totalCollateral()`, without checking

availableCollateral or the target LTV. This means the withdrawn amount per position is determined solely by its share of total collateral, regardless of how leveraged the position already is.

As a result, proportional withdrawals can push individual positions past the manager's target LTV, potentially re-leveraging them to the Morpho market's LLTV boundary. This undermines the LTV invariant that the position manager is designed to maintain and may leave positions closer to liquidation than intended.

Code corrected:

In Version 3 a new checkLtv parameter was added to proportional withdrawals that makes sure that the PositionManager LTV is being respected.

8.22 Repaying totalBorrowed() Can Revert Due to Rounding Mismatch

Correctness Low Version 2 Code Corrected

CS-GRUNT-097

MorphoBorrowPosition.repay() accepts an asset amount and forwards it to MORPHO.repay() with shares = 0, letting Morpho convert the asset amount back to shares internally. A natural usage pattern for full repayment is repay(totalBorrowed()), where totalBorrowed() returns the position's debt in asset terms.

totalBorrowed() converts the position's borrowShares to assets using toAssetsUp, which rounds up to ensure the reported debt is never understated. Morpho's repay() then converts the supplied asset amount back to shares using toSharesDown, which rounds down to protect the protocol. When these two roundings compose, the rounded-up asset amount can map back to a share count that exceeds the position's actual borrowShares. Morpho's checked subtraction (position.borrowShares -= repaidShares) then underflows and the transaction reverts.

This means a straightforward full repayment via repay(totalBorrowed()) can fail unexpectedly. This can happen as part of:

- rebalancing operations
- withdraw and burn from the PositionManager with sequential strategy, in _withdrawSequential
- theoretically also in _withdrawProportional

Code corrected:

In the new version totalBorrowed() uses toAssetsDown, rounding down the debt. This way, when the user calls repay(totalBorrowed()), the rounded-down asset amount will always correspond to a share count that is less than or equal to the actual borrowShares, preventing any underflow in the subtraction and ensuring the transaction succeeds as expected.

8.23 Reverting a Deposit Might Be Ineffective

Design Low Version 2 Risk Mitigated

CS-GRUNT-086

Version 2 allows the owner or a compliance role to revert a pending deposit by calling `FacilityLP.revertDeposit()`. This function burns the depositor's LP shares via `_burn(from, id, balance)`, triggering the `_beforeTokenTransfer` hook, which reverts if the depositor is not on the transfer allowlist:

```
if (!ITransferGuard(guard).canTransfer(_intent.properties.guardKey, from, to, amount)) {
  revert LibFacilityErrors.TransferBlocked(guard, from, to, amount);
}
```

A deposit from a blocked depositor therefore cannot be reverted.

On the other hand, if a deposit is reverted before the depositor is blocked, then nothing prevents the depositor from calling `deposit()` again while the intent is still in the depositing phase.

Risk mitigated:

3F has decided to mitigate the risk of this occurring with the following strategy:

```
There is no clean or elegant way to bypass the compliance check within the
ERC-6909 hook, including approaches based on transient storage.  As a result,
even if it seems counterintuitive, the process will require calling
revertDeposit before blocking an address.

In practice, this is acceptable because it will typically occur during a
phase when deposits are already disabled, which prevents users from immediately
redepositing after a revertDeposit.  The revertDeposit function itself is
expected to be used infrequently, and the added complexity in the flow is
considered an acceptable trade off to keep the overall implementation simpler.
```

8.24 Withdrawing availableCollateral Can Fail the Health Check

Correctness **Low** **Version 2** **Code Corrected**

CS-GRUNT-099

`MorphoBorrowPosition.availableCollateral(ltv)` computes the maximum collateral that can be withdrawn while keeping the position healthy at the given LTV. However, calling `withdraw(availableCollateral(ltv))` can revert because the subsequent `_isHealthy` check uses a different rounding strategy that may deem the position unhealthy.

`availableCollateral` delegates to `_requiredCollateral`, which inverts a single combined division: `borrowed * ORACLE_PRICE_SCALE / (ltv * price)` with one floor operation. The `_isHealthy` check applied after the withdrawal computes the collateral value using two separate floors:

1. `floor(collateral * price / ORACLE_PRICE_SCALE)`,
2. then `floor(result * ltv / WAD)`.

This can cause `_isHealthy` to undercount the position's collateral value by 1 unit compared to what `_requiredCollateral` assumed.

This is the same class of double-floor versus single-division rounding mismatch as the `preLiquidate` issue. The result is that maximum-withdrawal operations intermittently revert. This can happen as part of:

- rebalancing operations: when the rebalancer temporarily moves collateral and calls `withdraw(availableCollateral(safeLtv))`

- withdraw and burn from the `PositionManager` with sequential strategy, in `_withdrawSequential`, if `PositionManager.ltv == safeLTV`

Code corrected:

The computation of `availableCollateral(ltv)` was changed so that the available collateral can no longer be overestimated. Therefore, withdrawing the `availableCollateral(ltv)` no longer violates the `ltv`.

8.25 `maxBorrow` Returns an Unborrowable Amount

Correctness | Low | Version 2 | Code Corrected

CS-GRUNT-087

The `maxBorrow` view function computes the maximum additional amount the position can borrow at a given `LTV` by calculating `collateral * price * ltv - currentDebt`, where all intermediate terms use floor rounding (`mulDiv`, `mulWad`) and the current debt uses ceiling rounding (`toAssetsUp`). The `borrow` function relies on the same `_isHealthy` check that evaluates the post-borrow debt by converting the position's updated borrow shares back to assets with `toAssetsUp`. An integrator calling `borrow(maxBorrow(safeLtv))` therefore expects the transaction to succeed, since the view function advertises this value as the borrowable capacity.

However, when Morpho executes the borrow it converts the asset amount to shares via `toSharesUp` (rounding up), which encodes slightly more debt than the raw amount. After the market totals are updated with the new shares and assets, `_isHealthy` converts the position's total borrow shares back to assets using `toAssetsUp`, and the compounded upward rounding can produce a post-borrow debt that exceeds the floor-rounded capacity by one unit. This mismatch is most apparent in markets with an existing non-trivial borrow share exchange rate, where the virtual-offset arithmetic in `SharesMathLib` does not cancel out cleanly.

Borrowing the exact amount returned by `maxBorrow` can therefore revert with `InsufficientCollateral`, preventing the position from utilizing its full advertised capacity. The same inconsistency propagates to `borrowForCollateral`, which also delegates to `_maxBorrow`, and may cause automated rebalancing or leveraging strategies that rely on these view functions to fail unexpectedly.

Code corrected:

The new version of `_maxBorrow` simulates the asset-share-asset conversion process to account for the rounding effects in both directions. This ensures that the final asset amount corresponds to a value that does not exceed the allowed borrowing capacity, preventing the off-by-one rounding issue and unexpected reverts.

8.26 Burning All Shares Reverts With Depleted Principal Assets

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-014

The `burnAll` function in the `VaultController` allows users to burn all vault shares (PT and YT) and receive underlying assets.

`VaultController.burnAll` relies on `VaultController.convertToAssets` to determine the amount of underlying assets to return. When the contract has no principal asset, but a user possesses some shares, `VaultController.convertToAssets` incorrectly calculates the return amount to be equal to the user's `ptShares`, rather than zero.

```
function convertToAssets(uint256 ptShares, uint256 ytShares)
            public view returns (uint256 pAssets, uint256 yAssets) {
    (uint256 totalPAssets, uint256 totalYAssets, uint256 totalPtSupply,
                        uint256 totalYtSupply) = _assetsAndSupplies();
    pAssets = _eitherIsZero(totalPAssets, totalPtSupply)
        ? _initialConvertToAssets(ptShares, false)
        : ptShares.mulDiv(totalPAssets, totalPtSupply);
    yAssets = _eitherIsZero(totalYAssets, totalYtSupply)
        ? _initialConvertToAssets(ytShares, true)
        : ytShares.mulDiv(totalYAssets, totalYtSupply);
}

function _initialConvertToAssets(uint256 shares, bool yt) internal pure returns (uint256 assets) {
    assets = yt.ternary(0, shares);
}
```

This incorrect, non-zero amount is then used in `safeTransfer`. Since the contract has no assets to send, the transfer fails, and the entire transaction reverts. This might break integrations trying to burn all their remaining shares, however, it is also a fairly unlikely scenario.

Code corrected:

`VaultController.convertToAssets` falls back to `mulDiv` when the vault has been repaid instead of returning the 1:1 initial conversion.

```
bool repaid = _canWithdraw();
...
pAssets = (totalPtSupply == 0 || (totalPAssets == 0 && !repaid))
  ? _initialConvertToAssets(ptShares, false)
  : ptShares.mulDiv(totalPAssets, totalPtSupply);
```

Hence, when if `totalPAssets == 0`, `pAssets = 0` and a revert is avoided.

8.27 Code Duplication and Inconsistencies in Oracle Initialization

Design | Low | Version 1 | Code Corrected

CS-GRUNT-015

The USCCFund contract provides a `setOracle()` function that writes the oracle address to storage and emits an `OracleUpdated` event. The `initialize()` function performs the same oracle assignment and validation but writes directly to `_storage.oracle` rather than calling `setOracle()`.

This inconsistency means the initial oracle address set during deployment is never recorded via the `OracleUpdated` event. The validation logic (contract check and decimals check) is also duplicated inline in `initialize()` rather than reused from the helper.

Code corrected:

`USCCFund.initialize()` calls the shared `_setOracle(oracle_)` helper in Version 2.

8.28 Curator Can Insert Duplicate Entries in Supply and Withdrawal Queues

Correctness Low Version 1 Code Corrected

CS-GRUNT-058

The `setSupplyQueue` and `setWithdrawalQueue` functions in `PositionManagerAdmin` allow the curator role to configure the order in which borrow modules are used for deposits and withdrawals. Both functions validate that each entry is a whitelisted borrow module, but neither checks for duplicate entries within the queue.

A curator can therefore submit a queue containing the same module multiple times. For the supply queue, this causes deposits to be routed to the same module repeatedly. For the withdrawal queue, the same module is iterated over multiple times during withdrawal processing.

While not directly exploitable given the curator is a semi-trusted role, duplicate entries lead to redundant iterations and unexpected allocation behavior. A uniqueness check on queue entries would enforce the intended one-entry-per-module invariant.

Code corrected:

`PositionManagerAdmin.setSupplyQueue()` and `setWithdrawalQueue()` now include nested loop checks that detect duplicate entries.

8.29 Facility Transfers Zero-Amount Tokens When Share Proportion Rounds Down

Design Low Version 1 Code Corrected

CS-GRUNT-017

The `FacilityLP.claim()` function distributes tokens proportionally to burned shares using floor-division (`mulDiv`), iterating over every token held by the intent. When a user's proportional share of a particular token is fractionally small, `mulDiv` rounds `userBalance` down to zero.

The function does not check for a zero result before calling `transferTokenTo`, which forwards directly to Solady's `safeTransfer`. While most ERC-20 tokens treat zero-value transfers as no-ops, certain non-standard tokens (e.g., LEND) revert on zero-amount transfers, causing the entire `claim()` call to fail.

Because the loop transfers all tokens in a single transaction, a single zero-rounding revert on any one token blocks the user from claiming any of their tokens for that intent.

Code corrected:

`FacilityLP.claim()` now skips the transfer when the amount is zero, preventing reverts from non-standard tokens.

8.30 Front-Running Prime Broker Deposits

Trust | Low | Version 1 | Code Corrected

CS-GRUNT-018

The `Request` contract allows a privileged role (`OWNER` or `CONSUMER` role) to pre-authorize a funding transaction for a specific prime broker using `authorizeMinting`. Then, the prime broker calls the `Request.mint()` function, and thereby transfers the required amount of the underlying asset to the contract. Furthermore, the pre-authorized amounts of PT and YT are minted to the prime broker.

The front-running vulnerability exists because the direct authorization is split into two steps: the `Request.authorizeMinting()` call and the `Request.mint()` call. This separation makes it possible for a malicious or compromised owner/consumer to front-run a pending `Request.mint()` transaction, replacing a previously agreed-upon authorization with different terms.

As an example, a Prime Broker might have only intended to provide 1 million USDC, but as they had given an infinite USDC approval, their full USDC balance can be pulled into the `Request` contract and used as a bridge loan.

Version 2

`Request.mint()` has been updated to accept two additional arguments `minPt` and `minYt`. The function reverts when the pre-authorized amounts are less than expected.

```
(uint128 ptMintAuth, uint128 ytMintAuth) = msg.sender.mintAuth();
if (ptMintAuth < minPt || ytMintAuth < minYt) revert LibRequestErrors.SlippageExceeded();
msg.sender.updateMintAuth(0, 0);
_asset().safeTransferFrom(msg.sender, address(this), ptMintAuth);
```

This prevents the `CONSUMER` from front-running the call to `mint()` by lowering the `yt` amount. However, they can still provide a value that is larger than the minimum value provided. For `yt` this is not an issue as the prime broker receives it for free, however for `pt` this is problematic as the prime broker has to pay for the authorized amount. As described above, if the prime broker has granted an infinite USDC approval, a front-run increasing the `pt` authorization could still result in the prime broker's full USDC balance being pulled.

In that example the prime broker could provide a big bridge loan but only receive a small yield.

Code corrected:

`Request.mint()` now accepts a `maxPt` parameter instead of `minPt`. The function reverts if the pre-authorized PT amount exceeds `maxPt`, ensuring no more assets are transferred than intended by the caller:

```
if (ptMintAuth > maxPt || ytMintAuth < minYt) revert LibRequestErrors.SlippageExceeded();
 msg.sender.updateMintAuth(0, 0);
 _asset().safeTransferFrom(msg.sender, address(this), ptMintAuth);
```

8.31 Incomplete ERC-4626 Compliance for ControlledVault

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-019

`ControlledVault` implements the EIP-4626 Tokenized Vault Standard and restricts withdrawals and redemptions until the underlying bridge loans are repaid by the `Facility` or the deadline has passed.

Both `maxWithdraw` and `maxRedeem` correctly return zero when withdrawals are disabled. However, when withdrawals are enabled, both functions unconditionally return `type(uint256).max`, ignoring the `owner` parameter entirely. EIP-4626 requires these functions to return a value no higher than the amount that would actually be accepted for that specific owner address.

```
function maxWithdraw(address) external view returns (uint256) {
    return VaultController(_controller()).canWithdraw().ternary(type(uint256).max, 0);
}

function maxRedeem(address) external view returns (uint256 maxShares) {
    return VaultController(_controller()).canWithdraw().ternary(type(uint256).max, 0);
}
```

Code corrected:

`ControlledVault.maxWithdraw` and `maxRedeem` get the balance of the owner in Version 4.

```
function maxWithdraw(address owner) external view returns (uint256) {
    if (!VaultController(_controller()).canWithdraw()) return 0;
    return convertToAssets(this.balanceOf(owner));
}

function maxRedeem(address owner) external view returns (uint256) {
    if (!VaultController(_controller()).canWithdraw()) return 0;
    return this.balanceOf(owner);
}
```

8.32 Incorrect Event Emission During Rebalancing

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-020

The `Rebalanced` event is defined with three parameters: `positionManager`, `flashLoanAmount`, and `debtExcess` (the excess debt returned after rebalancing). The emission hardcodes `debtExcess` to 0, despite the function transferring actual excess debt tokens to the receiver via `debtAsset.safeTransferAll(receiver)`.

The `debtExcess` value is available (it is the return value of `safeTransferAll`) but is discarded. The hardcoded 0 contradicts the NatSpec definition, which describes `debtExcess` as "the excess debt returned from the rebalance,".

Off-chain systems relying on this event to reconcile flash loan economics will see every rebalance as perfectly balanced with zero excess, masking the true token flows.

Code corrected:

`PositionManagerRebalancing.rebalance()` now captures the return values of `collateralExcess` and `debtExcess` from the `safeTransferAll` calls. The `Rebalanced` event was renamed to `MorphoRebalanced` and updated to include both excess values.

8.33 Incorrect NatSpec Comment on resolveStart

Correctness Low Version 1 Code Corrected

CS-GRUNT-021

The `IntentProperties` struct in `LibIntent` includes a `resolveStart` field with a NatSpec comment describing it as the "Earliest timestamp when the intent can be resolved."

The comment is incorrect: `resolveStart` represents the latest timestamp at which the intent has to be resolved, not the earliest.

Code corrected:

The NatSpec has been corrected.

8.34 Maximum Management Fee Cap of 50% Is Excessively Permissive

Design Low Version 1 Code Corrected

CS-GRUNT-023

The `LibConstants` library defines `MAX_MANAGEMENT_FEE` as 5000 basis points, allowing the management fee to be set as high as 50% per year. This constant serves as the upper bound enforced when configuring fees on the Position Manager.

A 50% annual management fee far exceeds industry norms. A tighter upper bound would provide stronger guarantees to depositors that fees remain within reasonable expectations.

Code corrected:

`MAX_MANAGEMENT_FEE` in `LibConstants.sol` has been reduced to 200 basis points (2% per year).

8.35 Missing State Checks for setFund and setRequest

Correctness Low Version 1 Specification Changed

CS-GRUNT-024

The Facility's intent lifecycle is designed around a clear, three-stage process to ensure predictable and secure operations for LPs.

1. *DEPOSITING* Phase: The intent is open for capital. LPs can freely deposit and withdraw assets based on the intent's initial configuration.
2. *RESOLVING* Phase: This begins when either a facilitator manually calls `FacilityIntent.lock` or the pre-configured `resolveStart` deadline is reached. At this stage, all deposits and

withdrawals are blocked. The capital is locked in, and the facilitator performs the core operational duties, such as interacting with fund wrapper and position manager.

3. *RESOLVED* Phase: The final phase is initiated only when the facilitator calls `FacilityIntent.resolve`. This call can only succeed once all operational conditions are met, most critically, that any associated loan has been fully repaid. Once resolved, LPs can claim their pro-rata share of the resulting assets.

According to the provided specification, the `FacilityIntent.setFund` and `FacilityIntent.setRequest` functions, which define critical operational parameters, are intended to be executed exclusively during the `RESOLVING` phase. However, the implementation in `FacilityIntents` fails to enforce such state checks.

Specification changed:

The documentation was updated to reflect that `setFund`/`setRequest` should be callable in any intent phase by design.

8.36 Morpho Reference Can Be Immutable

Design | Low | Version 1 | Code Corrected

CS-GRUNT-066

The `morpho` field in `MorphoBorrowPosition` is stored in the ERC-7201 namespaced storage struct and read via `SLOAD` on every external call. Since the Morpho contract address is set once during `initialize()` and never changes, and since it is the same across all instances of `MorphoBorrowPosition`, it could be stored as an immutable variable instead.

Replacing the storage read with an immutable saves a warm `SLOAD` (2100 gas) on every operation that interacts with Morpho, which includes `supplyCollateral`, `withdrawCollateral`, `borrow`, `repay`, `preLiquidate`, and all view functions.

Code corrected:

The Morpho reference in `MorphoBorrowPosition.sol` is declared as immutable.

8.37 No PositionManager LLTV Check Enforced During Deposits

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-095

`LibOperations.processDeposit()` borrows against and distributes collateral to Morpho positions pro-rata without checking whether the resulting loan-to-value ratio exceeds the configured `PositionManager.lltv` threshold.

This is inconsistent with `LibOperations.processWithdrawals()`, which withdraws collateral while ensuring that each position's LTV stays below the configured `PositionManager.lltv`.

A deposit taking out a large amount of debt can therefore push a position's LTV above the configured limit.

Code corrected:

`processDeposit()` now queries `MorphoBorrowPosition.collateralForBorrow()` to determine how much collateral each position needs to reach the target LTV, accounting for existing collateral and debt. If the required collateral exceeds what is available, `borrowForCollateral()` reduces the borrow amount accordingly. Any leftover collateral not needed for borrowing is deposited as idle collateral into the first supply-queue position.

8.38 Off-by-one in Offer Expiration Check

Correctness Low Version 1 Code Corrected

CS-GRUNT-067

The `Offer` struct's NatSpec in `IOfferReceiver` defines `expiration` as the "Unix timestamp after which the offer becomes invalid." This implies the offer should still be valid at exactly `block.timestamp == expiration`.

However, `_validateOffer` in `OfferReceiver` uses `offer.expiration <= block.timestamp`, which reverts when the timestamp equals the expiration. This does not match the documented semantics.

Code corrected:

The check in `_validateOffer` was changed to `offer.expiration < block.timestamp`.

8.39 One Fund Simultaneously Assigned to Multiple Intents

Security Low Version 1 Code Corrected

CS-GRUNT-025

The `setFund` function in `FacilityIntents` (which can only be called by the facilitator) manages the association between intents and fund contracts, maintaining a reverse mapping (`fundsIntent`) to ensure each fund is only used by one intent. This mapping is checked via `checkFundIntent` and cleared via `abandonFund` when replacing a fund.

When `setFund(id, fundAddress)` is called twice with identical parameters, the second call passes the `checkFundIntent` check (since the fund is already mapped to the same intent), but then `abandonFund` clears the reverse mapping for the current fund. After the second call completes, the fund remains assigned to the intent but `fundsIntent[fund]` is deleted, leaving the reverse mapping empty.

With the reverse mapping cleared, the same fund can now be assigned to a different intent via another `setFund` call, bypassing the intended one-fund-per-intent invariant. This could allow a malicious or careless facilitator to bind a single fund to multiple intents simultaneously, potentially causing accounting errors or fund misallocation during order processing.

Code corrected:

`setFund()` in `FacilityIntents.sol` now skips state changes when the specified fund address is equal to the current one.

8.40 Paused Position Manager Allows Deposits and Withdrawals When totalAssets Is Unchanged

Correctness Low Version 1 Code Corrected

CS-GRUNT-027

The `PositionManager` enforces a pause check via the `TransferGuard` in `rebalance()`, but the `deposit()` and `withdraw()` functions in `PositionManagerLP` contain no pause check of their own. Instead, they rely on `_settleShares()` to mint or burn shares, which in turn triggers `_beforeTokenTransfer` where the `TransferGuard` blocks the transfer.

However, `_settleShares()` only calls `_mint` or `_burn` when `totalAssetsAfter` differs from `totalAssetsBefore`. When the two are equal (for example, a deposit of collateral paired with an equivalent debt extraction) `_settleShares` returns zero and neither mints nor burns, meaning `_beforeTokenTransfer` is never invoked and the pause check is never reached.

This allows a caller with `MINTER_ROLE` to execute deposit and withdraw operations that move tokens through the `PositionManager` even while the contract is paused, as long as the net asset impact is zero. The pause mechanism is intended to freeze all activity during incidents, and this bypass undermines that guarantee by permitting token flows that should be blocked.

Version 2:

`PositionManagerLP._settleShares()` now queries the transfer guard directly when net asset impact is zero, closing the bypass that allowed deposit and withdraw operations to proceed while the contract was paused.

However, due to other code changes the function `_mint()` and thereby `_beforeTokenTransfer` is never called when `totalAssetsAfter > totalAssetsBefore` but `convertToShares` rounds down to zero. Therefore, the issue remains.

Code corrected:

In Version 4, `PositionManagerLP._settleShares` explicitly queries `ITransferGuard.paused` when the shares to mint round to zero or when total assets are unchanged.

8.41 Performance Fees Charged on Gross Gain Instead of Net Gain

Design Low Version 1 Code Corrected

CS-GRUNT-028

The Position Manager charges performance fees whenever `totalAssets` increases relative to the previously saved value. However, the fee calculation uses the gross asset gain before management fees are deducted, rather than the net gain after accounting for them.

This means the performance fee base includes value that has already been claimed as management fees. If `totalAssets` grew by 50 assets but `managementFeeAssets` is 60 assets, performance fees are still charged on the full 50 asset gain even though the net position actually suffered a loss of 10 assets.

As a result, LPs are overcharged on performance fees in every period where management fees are non-trivial relative to asset growth. In extreme cases where management fees exceed the gross gain, performance fees are extracted despite the fund being net-negative for LPs.

Code corrected:

The `_pendingFees()` function in `PositionManagerBase.sol` now computes the management fee only on the net gain (`totalAssets_ - _lastTotalAssets - managementFeeAssets`).

8.42 PositionManager Deposits and Withdrawals Revert on Small Asset Deltas

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-029

The `_settleShares()` function in `PositionManagerLP` computes the shares to mint or burn by converting the asset delta (`totalAssetsAfter - totalAssetsBefore`) via `convertToShares`, which uses `mulDiv` with virtual offsets. When the asset delta is non-zero but very small relative to the total assets, this conversion rounds down to zero. Then, the function explicitly reverts with `ZeroShares()`.

This can occur in practice because `totalAssets()` is derived from oracle-priced collateral minus debt (`collateralAmountQuoted - debtAmount`), meaning small oracle price movements or Morpho interest accrual between the `_accrueFees()` snapshot and the post-operation `totalAssets()` call can introduce tiny, unintended asset deltas. A transaction that succeeded in simulation might fail on-chain.

The result is that legitimate operations (particularly those with balanced collateral/debt movements) can theoretically fail even though it is very unlikely due to the virtual shares used.

Code corrected:

`PositionManagerLP._settleShares()` now handles the asset-increase path by skipping the mint when `convertToShares` rounds down to zero, treating dust-level deltas as a no-op rather than reverting. The asset-decrease path rounds up when computing `sharesToBurn`, ensuring a non-zero asset decrease always produces at least one share to burn.

8.43 Prime Broker Can Front-Run authorizeMinting to Double-Mint

Security | Low | Version 1 | Risk Mitigated

CS-GRUNT-068

The `authorizeMinting` function in `Request` sets the mint authorization to the specified amount, overwriting any previous value. The authorized party can then call `mint()` to consume the full authorization. This pattern is vulnerable to the same front-running issue as ERC-20's `approve`.

Consider the following scenario:

1. The consumer calls `authorizeMinting(primeBroker, 1M, 0)` to authorize minting 1M PT.
2. The prime broker requests a higher allocation; the consumer agrees to 2M.
3. The consumer calls `authorizeMinting(primeBroker, 2M, 0)` to update the authorization.
4. The prime broker front-runs this transaction by calling `mint()`, consuming the original 1M authorization.

5. After the second `authorizeMinting()` is executed, the prime broker calls `mint()` again, consuming the new 2M authorization.

The prime broker receives 3M PT in total instead of the intended 2M. Since `authorizeMinting` unconditionally overwrites the stored value rather than adjusting it, there is no way for the consumer to safely update an outstanding authorization without first coordinating with the authorized party off-chain or setting the authorization to zero and waiting for confirmation before setting the new value.

Risk Mitigated:

3F acknowledged the risk and provided the following mitigation strategy:

```
The revoke path (call authorizeMinting with 0, 0) overwrites the broker's
current auth slot to zero. To safely update: first revoke to 0, wait for
confirmation, then set new authorization. No old authorization remains to
combine with the new one.
```

8.44 Redundant Deletions in USCCFund

Design | Low | Version 1 | Code Corrected

CS-GRUNT-030

The `USCCFund` deletes `_storage.resolvedOrder` variable in four places: create(), cancel(), recover(), and unlock(). The `resolvedOrder` is only written by the `resolve()` function and should be kept until the `ENDED` state is reached.

In case `resolvedOrder` has been set, the subsequent create() call, which is the only path to reuse the storage, already clears `resolvedOrder`.

Each unnecessary delete on a struct with multiple slots costs additional gas per slot.

Code corrected:

The `resolvedOrder` struct in `UsccFundStorage` has been replaced with state variables (`resolvedInput`, `resolvedOutput`, `hasResolvedAmounts`). Only `create()` resets the fields; all other deletions are removed.

8.45 Redundant Storage in USCCFund

Design | Low | Version 1 | Code Corrected

CS-GRUNT-031

The `UsccFundStorage` struct persists the full `Order` struct in `currentOrder` alongside its hash in `currentOrderId`. Every external function (cancel, commit, unlock, recover) receives the `Order` as a `calldata` parameter, recomputes its hash via `order.toId()`, and validates it against the stored `currentOrderId`. None of these functions ever read `_storage.currentOrder`.

Writing a full `Order` struct (mode, owner, receiver, two `uint256`, and a `bytes32` salt) to storage on every create() and clearing it on cancel() costs multiple SSTOREs per lifecycle.

Code corrected:

The `currentOrder` struct has been removed from `UsccFundStorage`.

8.46 Role Constants Are Public in USCCFund and TransferGuard

Design | Low | Version 1 | Code Corrected

CS-GRUNT-070

In `USCCFund`, the role constants are declared as `public`:

```
uint256 public constant OPERATOR_ROLE = _ROLE_0;
uint256 public constant DEPOSITOR_ROLE = _ROLE_1;
```

Similarly, in `TransferGuard`:

```
uint256 public constant COMPLIANCE_ROLE = _ROLE_0;
uint256 public constant PAUSER_ROLE = _ROLE_1;
```

All other contracts in the codebase declare their role constants as `internal`, for example in `FacilityRoles`, `PositionManagerBase`, `Request`, and `WrappedAsset`. The `public` visibility generates automatic getter functions that are unlikely to be needed on-chain, as the role values are compile-time constants that off-chain integrators can read from the source.

This creates unnecessary getters, increasing the bytecode size and hence increasing the deployment gas cost.

Code corrected:

The aforementioned role constants are now declared as `internal`.

8.47 Target PM Restrictions Unchecked When Deposit PM Is Guard Key

Correctness | Low | Version 1 | Risk Mitigated

CS-GRUNT-033

`FacilityIntent` supports PM-to-PM intents where both the deposit and target assets are position managers, allowing users to migrate between leveraged positions. In this configuration, `LibIntent._checkAssetsAndGuardKey()` permits either position manager to be designated as the intent's guard key, provided both share the same collateral and debt assets.

A guard key governs access to an intent — a position manager acting as guard key enforces its own transfer restrictions, such as whitelists or blocklists, on all intent participants. When the deposit PM is selected as the guard key, only its access restrictions gate access to the intent; the target PM's transfer restrictions are never evaluated.

```
} else if (depositIsPm && targetIsPm) {
    if (guardKey != depositAsset.asset && guardKey != targetAsset.asset) {
        revert LibFacilityErrors.InvalidGuardKey(guardKey);
    }
    // No check on the target PM's own transfer restrictions
```

```
    ...
}
```

A user who passes the deposit PM's access controls but is blocked by the target PM's restrictions can successfully deposit into the intent. Yet they will be unable to receive the target PM shares upon settlement. The deposited funds become stuck inside the intent with no path for recovery.

Risk mitigated:

3F stated:

```
If an intent has 2 PMs, they share the same TransferGuard.
This is not enforced on-chain but will be managed operationally.
```

8.48 TokenController Reverts on Self-Transfers Deviating From ERC-20 Standard

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-034

The `TokenController._transfer` function includes an explicit check that reverts when the sender and recipient are the same address. The ERC-20 standard does not prohibit self-transfers, and widely used implementations such as OpenZeppelin treat them as valid no-ops.

This non-standard restriction introduces an unexpected revert path that external integrators have no reason to anticipate. Smart contract flows, routers, or aggregators that route tokens through intermediate steps may legitimately produce self-transfers as an edge case.

As a result, composability with external protocols is reduced, as any integration that triggers a self-transfer will fail silently at the token level. This can cause hard-to-diagnose issues in multi-step transactions and discourage protocol integrations.

Code corrected:

The `TokenController._transfer()` allows self transfers.

8.49 USCCFund Sends Assets to Msg.Sender Instead of Order.Receiver

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-035

The `Order` struct defines a receiver field intended to designate the address that receives the output of an operation, separate from the owner who initiates it. The `USCCFund` functions `unlock()` and `recover()` both transfer output assets (USDC or wUSCC) to `msg.sender` rather than to `order.receiver`.

The `create()` function enforces `order.receiver == msg.sender`, so in practice the two addresses are always equal and no funds are currently misdirected. However, the code contradicts the `Order` struct's design intent. If the receiver check in `create()` were ever relaxed to support third-party receivers, the fund operations would silently send assets to the wrong address.

Code corrected:

Functions `unlock()` and `recover()` in `USCCFund.create()` were updated to send assets to the `order.receiver`.

8.50 WrappedAsset Initialization Allows Decimal Mismatch

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-039

The Grunt protocol uses `WrappedAsset` as a 1:1 wrapper for receipt tokens from asynchronous settlements with funds (e.g., Superstate's USCC). The primary purpose of this wrapper is to control where the receipt token can be used as collateral. However, the `initialize` function allows a deployer to set the `decimals` parameter without validating it against the decimals of the `underlying` token. Therefore, the 1:1 peg assumption might not hold from a user perspective.

```
function initialize(
    address owner_,
    address initialIssuer_,
    address underlying_,
    string calldata symbol_,
    string calldata name_,
    uint8 decimals_
) public initializer {
    WrappedAssetStorage storage _storage = _wrappedAssetStorage();
    _storage.symbol = symbol_;
    _storage.name = name_;
    _storage.decimals = decimals_;
    _storage.underlying = underlying_;

    _initializeOwner(owner_);
    _setRoles(initialIssuer_, ISSUER_ROLE | SENDER_ROLE);
}
```

Code corrected:

The `WrappedAsset.decimals()` function derives the decimals from the underlying token as `ERC20(underlying).decimals()`.

8.51 _settleShares Rounds Down Shares Burned

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-040

The `_settleShares` function in `PositionManagerLP` computes how many shares to burn from the active user when a deposit or withdrawal operation causes `totalAssets` to decrease. It delegates to `LibView.convertToShares`, which unconditionally uses `mulDiv` (round down). Because the caller is

losing shares, the correct rounding direction is up. The caller should surrender at least as many shares as the value they extracted:

```
function _settleShares(uint256 totalAssetsBefore, uint256 _totalSupply) internal returns (int256 sharesDelta) {
    PositionManagerStorageData storage _storage = LibStorage.positionManagerStorage();
    uint256 totalAssetsAfter = _storage.totalAssets();

    if (totalAssetsAfter > totalAssetsBefore) {
      // Assets increased: mint shares to caller
      uint256 assetsAdded = totalAssetsAfter - totalAssetsBefore;
      uint256 sharesToMint = assetsAdded.convertToShares(_totalSupply, totalAssetsBefore); // rounds down
      ...
    } else if (totalAssetsAfter < totalAssetsBefore) {
      // Assets decreased: burn shares from caller
      uint256 assetsRemoved = totalAssetsBefore - totalAssetsAfter;
      uint256 sharesToBurn = assetsRemoved.convertToShares(_totalSupply, totalAssetsBefore); // rounds down
      ...
    }
}
```

Each withdrawal or net-negative deposit burns up to 1 wei fewer shares than the value removed. Over many operations this leaks fractional value from Position Manager LPs. Assuming a proper setup, this is economically negligible, but accumulates directionally against LPs across every such operation.

Code corrected:

The `_settleShares()` function in `PositionManagerLP.sol` rounds up the number of shares burned in Version 2.

8.52 addBorrowModule and removeBorrowModule Skip Fee Accounting

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-041

The `addBorrowModule` and `removeBorrowModule` functions can change the effective `totalAssets` of the Position Manager by adding or removing a module whose debt balance contributes to the total. However, neither function triggers the fee accounting update that normally reconciles management and performance fees before a change in total assets.

This means the fee state can become stale across a module change. Adding a module with existing funds inflates `totalAssets` without first checkpointing, potentially crediting unearned performance fees. Removing a module deflates `totalAssets`, potentially skipping fees that should have been accrued.

The practical likelihood is limited because both functions are restricted to `onlyOwner`. Nonetheless, an owner action that should be purely administrative can silently distort fee accounting for LPs.

Code corrected:

Both functions now accrue fees before changing the effective `totalAssets`.

8.53 repaymentDeadline Not Validated

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-042

The `Request.initialize()` function stores the `repaymentDeadline_` parameter directly into storage without checking it. This deadline controls when withdrawals are automatically enabled via `_syncWithdrawalStatus()`, which sets `repaid = true` once `block.timestamp >= repaymentDeadline`.

A past or current timestamp passed as `repaymentDeadline_` would cause the `Request` to be immediately in a repaid state from the moment of deployment.

A timestamp extremely far in the future effectively disables the functionality of the `repaymentDeadline`.

Code corrected:

`Request.initialize()` validates that `repaymentDeadline` is after the current block timestamp.

8.54 setRequest Can Bind the Same Request to Multiple Intents

Correctness | Low | Version 1 | Code Corrected

CS-GRUNT-043

The `setRequest` function in `FacilityIntents` maintains a reverse mapping to ensure each request is bound to only one intent. However, calling `setRequest` on the same intent with its already-assigned request first abandons the old mapping and then re-registers it, clearing the uniqueness check without actually changing the binding.

After this self-reassignment, the reverse mapping slot for that request is freed. A subsequent call to `setRequest` on a different intent can then bind the same request address, bypassing the `checkRequestIntent` guard.

This allows multiple intents to reference the same request contract, breaking the one-to-one invariant the mapping is designed to enforce. The impact is limited because `checkRequestRepaid` requires the existing request to be fully repaid before reassignment, restricting the window in which the duplicate binding can occur.

Code corrected:

`setRequest()` in `FacilityIntents.sol` now returns without effect when attempting to re-register a request.

8.55 Incorrect Comment About Fee Shares

Informational | Version 2 | Code Corrected

CS-GRUNT-089

Code comments in `MorphoBorrowPosition.availableLiquidity()` imply that fee shares minted to Morpho's fee recipient can influence the available liquidity:

```
// ... However, fee shares redirect a portion of supply,
// which can subtly affect the calculation. Using expected values is correct.
```

Fee shares only redistribute supply assets; they do not change `totalSupplyAssets` and therefore have no effect on the available liquidity.

Code corrected:

The misleading comment about fee shares was removed.

8.56 Misnamed Custom Error

Informational | Version 2 | Code Corrected

CS-GRUNT-090

The function `LibStorage.checkDigest()` reverts with the custom error `SwapDigestUsed()` when a digest has already been consumed. This name is misleading, as digests are also used when setting a new fund or request contract in Version 2.

Code corrected:

The custom error was renamed to `DigestUsed`.

8.57 Pause Duration Is One Second Longer Than Specified

Informational | Version 2 | Code Corrected

CS-GRUNT-091

`LibPause` provides pausable functionality where `pauseFor(duration)` computes a `pauseUntil` timestamp as:

```
block.timestamp + duration
```

and `paused()` checks `block.timestamp <= self` to determine whether the contract is paused.

Because the comparison is inclusive (<= rather than <), a call to `pauseFor(3600)` at timestamp `T` keeps the contract paused for timestamps `T` through `T + 3600` inclusive. Meaning, 3601 seconds total rather than the 3600 specified. This is a consistent off-by-one across all pause usage since `paused()` is the sole entry point for pause checks.

The practical impact is negligible.

Code corrected:

`LibPause.paused()` has been updated to use a strict less-than comparison <. Hence, the finding is resolved as the check is no longer inclusive.

8.58 Redundant Calls to Accrue Interest

Informational | Version 2 | Code Corrected

CS-GRUNT-092

The functions `MorphoBorrowPosition.withdrawCollateral()` and `borrow()` explicitly call `MORPHO.accrueInterest()` before calling the `withdrawCollateral()` and `borrow()` functions of Morpho respectively. Both of these Morpho operations internally accrue interest as their first step. This makes the preceding `accrueInterest` calls redundant and hence wastes some gas.

Code corrected:

The redundant calls to `accrueInterest()` were removed in Version 4.

8.59 Unchecked Return Value in `removeBorrowModule`

Informational | Version 2 | Code Corrected

CS-GRUNT-093

The function `PositionManagerAdmin.removeBorrowModule()` does not check the return value of `_storage.borrowModules.remove(module)`.

If the supplied address (which should be removed) is not in the borrow module set, `remove` silently returns `false` and the function proceeds to call `transferOwnership` on an arbitrary contract.

Code corrected:

The function `PositionManagerAdmin.removeBorrowModule()` checks the return value of `borrowModules.remove()` in Version 4.

8.60 Unreachable ZeroShares Revert in `_settleShares`

Informational | Version 2 | Code Corrected

CS-GRUNT-101

In `PositionManagerLP._settleShares()`, the branch handling decreasing assets contains a guard against zero shares:

```
uint256 sharesToBurn = assetsRemoved.convertToShares(_totalSupply,
                            totalAssetsBefore, virtualShareOffset_, true);
if (sharesToBurn == 0) revert LibManagerErrors.ZeroShares();
```

This revert is unreachable. The branch is only entered when `totalAssetsAfter < totalAssetsBefore`, guaranteeing `assetsRemoved > 0`. The `convertToShares` call passes `roundUp = true`, which uses `mulDivUp`. Since `assetsRemoved >= 1` and the numerator includes `_totalSupply + virtualShareOffset_` (which is always >= 1), `mulDivUp` is guaranteed to return at least 1.

The dead code is harmless. It adds a tiny gas overhead.

Code corrected:

The unreachable `ZeroShares` revert was removed in Version 4.

8.61 Arbitrary Borrower Address in `preLiquidate`

Informational | Version 1 | Code Corrected

CS-GRUNT-060

The `MorphoBorrowPosition.preLiquidate()` function accepts an arbitrary borrower address as a parameter, for whom debt is repaid and collateral is withdrawn.

Functionally, only the `MorphoBorrowPosition` contract itself should be subject to preliquidation. While this cannot be exploited to steal funds, it allows the caller to trigger a callback from Morpho into the `MorphoBorrowPosition` with no real use case and in an unexpected context.

Code corrected:

`MorphoBorrowPosition.preLiquidate()` now validates that the `borrower` parameter equals `address(this)`.

8.62 Arbitrary Decimals in PositionManager

Informational | Version 1 | Code Corrected

CS-GRUNT-061

During initialization of the `PositionManager` arbitrary values for `decimals` can be set.

But the initial share-to-asset conversion ratio is `VIRTUAL_SHARES = 1e6`, so the shares have decimals equal to debt token decimals plus six.

A mismatch between the owner-supplied `decimals` and the effective precision derived from the virtual offset could mislead integrators and front ends that rely on `decimals()` to interpret share balances.

Code corrected

The initial share-to-asset conversion has been changed. It now computes `virtualShareOffset = 10 ** max(18 - debtAsset.decimals, 0)`. `virtualShareOffset` is then used in the `convertToShares` function. Thereby, shares have 18 decimals. Additionally, the `decimals()` override has been removed, so the contract returns the Solady ERC20 default value of 18.

8.63 Inconsistent Validation for Order's Input Amount

Informational | Version 1 | Code Corrected

CS-GRUNT-045

The protocol enforces inconsistent validation for an order's `input` value at different stages of its lifecycle. The `FacilityFunds.create` function requires `order.input` to be non-zero upon creation. However, this non-zero check is absent in the `resolve` function.

Code corrected:

A non-zero check for `input` has been added to `resolve`.

8.64 Incorrect Error Referenced

Informational | Version 1 | Code Corrected

CS-GRUNT-073

The NatSpec documentation for `PositionManagerLP.deposit()` and `withdraw()` states that these functions revert with `{LibManagerErrors.ZeroAmount}`. However, the actual revert uses `CommonErrors.AmountZero()` from `LibCommonErrors`.

Further the `Paused` error is defined in `LibCommonErrors` rather than `LibManagerErrors`, as stated in `PositionManagerRebalancing.rebalance()`.

Code corrected:

The NatSpec documentation has been corrected.

8.65 Incorrect Return Values in USCCFund

Informational | Version 1 | Code Corrected

CS-GRUNT-046

The `IFund` interface specifies that `maxDeposit` returns the maximum amount of assets depositable by an account and that `maxRedeem` returns the maximum number of shares redeemable by an account.

`USCCFund.maxDeposit()` returns `type(uint256).max` despite the fund accepting no deposits for all users other than ones with the `_DEPOSITOR_ROLE`. Similarly, `USCCFund.maxRedeem()` returns `WUSCC.balanceOf(account)` despite the fund allowing no direct redemptions.

Code corrected:

`maxDeposit()` and `maxRedeem()` return zero for accounts without the `_DEPOSITOR_ROLE`.

8.66 Incorrect Safety Comment in TokenController

Informational | Version 1 | Code Corrected

CS-GRUNT-075

The internal `_transfer` function in `TokenController` contains the comment:

```
// casting to 'uint128' is safe because [The allowance is checked if higher than a 128 bit number]
```

This function does not deal with allowances; it operates on balances and verifies them against `ptBalanceSender` and `ytBalanceSender`.

The comment appears to have been copied from `_consumeAllowance`.

Code corrected:

The safety comment has been corrected.

8.67 Initialization Not Disabled in Implementation Contracts

Informational | Version 1 | Code Corrected

CS-GRUNT-047

Upgradeable contracts such as `Facility` use an `initialize` function gated by the `initializer` modifier to set up ownership and roles through a proxy. However, the implementation contracts themselves do not call `_disableInitializers()` in their constructor to prevent direct initialization on the logic contract.

This allows an attacker to call `initialize` on the unproxied implementation contract and claim ownership of it. While this does not directly compromise the proxy's state, the attacker could use the initialized implementation to execute unexpected calls through any logic that references `address(this)`.

Code corrected:

Upgradeable implementation contracts now call `_disableInitializers()` in their constructors.

8.68 Misleading Error State in USCCFund Revert Messages

Informational | Version 1 | Code Corrected

CS-GRUNT-048

The `recover()` and `unlock()` functions validate the order's current phase by calling `_state(order)`, which computes a dynamic state derived from on-chain balance checks — this can differ from the raw `_storage.internalState`.

When validation fails, both functions revert with `InvalidState(_storage.internalState)` instead of `InvalidState(_currentState)`. This means the error reports the raw internal state rather than the computed dynamic state that actually caused the check to fail.

Code corrected:

The `recover()` and `unlock()` functions in `USCCFund` return the dynamic state value in the revert.

8.69 Missing Asset Compatibility Check in MorphoBorrowPosition Initialize

Informational | Version 1 | Code Corrected

CS-GRUNT-076

The `initialize` function in `MorphoBorrowPosition` validates the market ID, LLTV bounds, and market existence but does not verify that the Morpho market's collateral and loan tokens match the collateral and debt assets expected by the owning `PositionManager`. A misconfigured initialization could silently attach a borrow position whose market operates on different tokens than the `PositionManager` expects.

Adding checks that `marketParams.collateralToken` and `marketParams.loanToken` match the `PositionManager`'s configured assets would prevent misconfiguration at deployment time rather than failing at runtime during the first collateral supply or borrow operation.

Code corrected:

`MorphoBorrowPosition.initialize()` now compares the collateral and debt asset of the position manager with the collateral and loan token on the morpho market.

8.70 Missing Function to Query Accurate Share Price

Informational | Version 1 | Code Corrected

CS-GRUNT-094

Comments in `IPositionManager` state that `totalAssets` can be used to compute the price of Position Manager shares. However an accurate share price requires both an up-to-date total assets value as well as an up to date value of the total supply. Management and performance fees are charged by minting Position Manager shares. As such the current `totalSupply()` does not include any pending fees. So integrators are not able to easily compute an accurate share price.

Code corrected:

A `pendingFees()` public view function has been added to `PositionManager.sol` which returns the pending management and performance fee shares that would be minted if `accrueFees()` were

called at the current block timestamp.

8.71 Missing Sanity Checks on LLTV

Informational | Version 1 | Code Corrected

CS-GRUNT-049

The `PositionManager` LLTV is not validated to be strictly smaller than the Morpho Borrow Position safe LLTV. If the `PositionManager` LLTV exceeds the safe LLTV, calling `PositionManager.withdraw` could push the position past the safe LLTV threshold, leading to reverts on `withdrawCollateral()`. Hence, such a combination of `PositionManager` and Borrow Position is essentially incompatible and hence should not be allowed.

Code corrected:

`PositionManagerAdmin.addBorrowModule()` now validates that the module's `safeLtv` is at least as large as the `PositionManager`'s target LTV. Additionally, `setLtv` checks that the `safeLtv` of all borrow modules is at least as big as the newly set LTV.

8.72 Missing Verification of Target Order

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

CS-GRUNT-077

The `recovering()` function in `USCCFund` transitions the internal state from `PROCESSING` to `RECOVERING` but does not accept an `Order` parameter or validate the `currentOrderId`. All other state-mutating functions (`commit`, `cancel`, `unlock`, `recover`) require the caller to pass the order and verify it matches `currentOrderId`.

If an operator's `recovering()` transaction remains pending in the mempool while the current order completes and a new order enters `PROCESSING`, the stale transaction will put the wrong order into the `RECOVERING` state.

Code corrected:

The `recovering()` function now accepts an `orderId` parameter and reverts if it does not match `currentOrderId`.

8.73 Missing Zero-Address Check in Claim and Withdraw

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

CS-GRUNT-078

The `claim` and `withdraw` functions in `FacilityLP` accept a `receiver` parameter but do not validate it against `address(0)`. Solady's `SafeTransferLib.safeTransfer` does not revert on transfers to the zero address. So, a user passing `address(0)` would permanently lose the withdrawn tokens.

This is self-griefing only, as the caller can only harm their own funds causing accidental token loss from misuse or faulty integrations.

Code corrected:

`FacilityLP.sol` now reverts in the functions `claim` and `withdraw` when `receiver` is `address(0)`.

8.74 No Upper Bound on setMaxRebalanceLoss

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

CS-GRUNT-050

The maximum rebalance loss controls the relative value that can be lost in a rebalancing event. The value is expressed in basis points so `10_000` represents 100%. The function `PositionManagerAdmin.setMaxRebalanceLoss()` accepts any `uint16` without verifying that the value is in the correct range. If the value is accidentally set too high all assets could be lost in a rebalancing.

Code corrected:

Version 2 of the code caps the possible rebalance loss by `MAX_REBALANCE_LOSS` in `LibStorage.setRebalanceConfig()` (set to 1000 basis points).

8.75 Outdated NatSpec Documentation

Informational | Version 1 | Code Corrected

CS-GRUNT-052

The codebase contains outdated documentation:

1. The lifecycle NatSpec in `Request.sol` states "the owner pulls funds … via `pullFunds()`, while in practice `pullFunds()` is restricted to `_ROLE_PULLER`, held by `Facility` contract.
2. The README refers to the `USCCFund` operator role as `Settler`, which does not match the role name used in the contract `OPERATOR_ROLE`.

Code corrected:

The NatSpec in `Request.sol` and the `README` have been updated to reflect the implemented code.

8.76 Solidity Version Not Fixed

Informational | Version 1 | Code Corrected

CS-GRUNT-053

The Solidity compiler version is not explicitly fixed. Neither the contracts nor the `foundry.toml` configuration specify a fixed compiler version. This can lead to inconsistencies in builds over time, future compiler versions may introduce subtle changes in behavior, and result in unexpected runtime behavior. This may also lead to tests being executed on a bytecode that does not match the deployed bytecode.

Code corrected:

The Solidity compiler version has been locked to `0.8.34` in `foundry.toml`.

8.77 Typo in _intialPmParameters Variable

Informational | Version 1 | Code Corrected

CS-GRUNT-054

The private function `_intialPmParameters` in `FacilityPositionManager` is missing the letter "i" in "initial" making it read "intial" instead.

Code corrected:

The function `_intialPmParameters` in `FacilityPositionManager.sol` has been renamed to `_initialPmParameters`.

8.78 USCCFund Uses Potentially Stale Superstate Allowlist Interface

Informational | Version 1 | Code Corrected

CS-GRUNT-079

`USCCFund.create()` checks permissions by resolving the allowlist from the USCC token and calling it through the project's own `IAllowlist` interface:

```
IAllowlist(ISuperstateToken(USCC).allowlistV2())
    .isAddressAllowedForPrivateInstrument(address(this), "USCC")
```

`SuperstateToken.isAllowed(address)` performs the equivalent check internally:

1. It calls `allowlistV2.isAddressAllowedForFund(addr, symbol())`
2. Which calls `isAddressAllowedForPrivateInstrument` in the Allowlist contract

Replacing the two-hop call with `ISuperstateToken(USCC).isAllowed(address(this))` would remove the dependency on the selector and make the fund more robust for future changes.

Code corrected:

`USCCFund.create()` calls `ISuperstateToken.isAllowed(address)` directly in Version 2.

8.79 Unnecessary Free Memory Pointer Update

Informational | Version 1 | Code Corrected

CS-GRUNT-080

The `LibOrder.toId()` function uses an assembly block marked `"memory-safe"` to compute a keccak256 hash of the order data. At the end of the block, it updates the free memory pointer:

```
assembly ("memory-safe") {
    let start := mload(0x40)
    mstore(start, chainid())
    mstore(add(start, 0x20), fund)
    mcopy(add(start, 0x40), order, 0xC0)
    id := keccak256(start, 0x100)
    mstore(0x40, add(start, 0x100))
}
```

Per the Solidity documentation, memory-safe assembly blocks may use *"temporary memory that is located after the value of the free memory pointer at the beginning of the assembly block, i.e. memory that is 'allocated' at the free memory pointer without updating the free memory pointer."*

Since the written memory at `start` through `start + 0x100` is only used for the `keccak256` computation and is not referenced after the assembly block, the free memory pointer update at the last line expands memory unnecessarily.

Code corrected:

The `LibOrder.toId()` function no longer updates the free memory pointer.

8.80 Unnecessary cachedBalance in USCCFund Logic

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

CS-GRUNT-056

The `USCCFund` contract retains a `cachedBalance` variable and its associated logic. This is a legacy design choice that is no longer used. Its continued presence introduces unnecessary code complexity and increased gas costs due to redundant storage operations.

Code corrected:

The `cachedBalance` was removed from the `USCCFund`.

8.81 addBorrowModule Lacks Validation

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

CS-GRUNT-057

The `PositionManager` maintains a set of `borrowModules`, which are `MorphoBorrowPosition` contracts, whose collateral and debt are aggregated into `totalAssets` for share pricing and fee calculations. The `addBorrowModule` function inside `PositionManagerAdmin` allows the owner to register any address as a borrow module without any checks.

`addBorrowModule` does not verify that the module's `owner()` is the calling `PositionManager`, nor that its collateral/debt assets match the `PositionManager`'s configured assets. This means a borrow position owned by a different `PositionManager`, or one operating on entirely different token markets, can accidentally be added. While the owner of the `PositionManager` is a trusted role, this seems like a credible mistake that could be avoided.

Mutative operations (supply, borrow, repay, withdraw) on the mismatched module would revert due to `onlyOwner` checks on the `MorphoBorrowPosition`, breaking deposit/withdrawal/burn/rebalance flows inside the `PositionManager` that iterate the queues. Additionally, view functions like `totalCollateral()` and `totalBorrowed()` would include values from the foreign position, inflating `totalAssets` and causing the fee accrual logic in `_accrueFees` to mint incorrect performance fee shares based on assets the `PositionManager` does not actually control.

Code corrected:

`PositionManagerAdmin.addBorrowModule()` now validates that the module's collateral and debt assets match the `PositionManager`'s and the module's `owner()` is the calling `PositionManager`.

9 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.

9.1 Conservative Rounding Propagates Minor Errors

Informational | Version 4 | Acknowledged

CS-GRUNT-103

`MorphoBorrowPosition.totalBorrowed()` uses `toAssetsDown` to convert borrow shares to asset amounts, rounding down to ensure `repay(totalBorrowed())` never reverts on Morpho. This conservative rounding was introduced deliberately in the latest version, but it causes `totalBorrowed()` to understate the true debt by up to 1 wei per borrow module.

This underestimation propagates through two paths in the `PositionManager`. First, `LibView.totalAssets()` computes net asset value as `collateral - debt` per module, so understated debt leads to a slightly overstated `totalAssets()`. This feeds into share-to-asset conversions via `convertToShares()` and into fee accrual via `_pendingFees()`, introducing rounding errors of at most 1 wei per module in both share minting/burning and management/performance fee calculations.

Second, `LibView.debtAmount()` (which sums `totalBorrowed()` across all modules) is used in `burn()` to compute the proportional debt a departing shareholder must repay. Because debt is understated, the burned user repays a tiny amount too little, leaving residual debt in the system and marginally decreasing the share price for remaining holders.

All errors are bounded to at most 1 wei per borrow module per operation and are economically negligible given the restrictions on supported ERC-20 tokens. Furthermore, all the relevant state functions can only be triggered by authorized roles.

Acknowledged:

The 3F has acknowledged the issue, but has decided to keep the code unchanged.

9.2 Full Withdrawal and Burn Can Revert Due to Rounding

Informational | Version 4 | Acknowledged

CS-GRUNT-104

The `PositionManager` allows withdrawals via `withdraw(collateral, debt, strategy)` with explicit amounts and exits via `burn(shares, strategy)` where amounts are derived proportionally. Both paths delegate to `processWithdrawal` which distributes repayment and collateral withdrawal across the queue using either a sequential or proportional strategy.

Because `totalBorrowed()` rounds down via `toAssetsDown`, the values returned by `collateralAmount()` and `debtAmount()` can be slightly inconsistent with what Morpho's internal accounting actually requires. This creates edge cases where a caller requesting seemingly valid "full exit" amounts triggers unexpected reverts.

For **withdrawals** with `collateral=collateralAmount()` and `debt=debtAmount()`, the sequential strategy can revert with `InsufficientAvailableCollateral` because the `repay` operations leave behind dust debt and then `availableCollateral()` computes available amounts conservatively and based on the dust debt. The proportional strategy can similarly revert with `InsufficientAvailableCollateral` at the per-position LTV check because after proportionally distributing the rounded-down debt, individual positions might retain dust debt but zero excess collateral, violating the health constraint.

For **burns** with `shares=totalSupply()`, the proportional amounts computed in `burn()` (collateral rounded down, debt rounded up via `mulDivUp`) can fail. The sequential strategy fails with `InsufficientAvailableCollateral` at the end when the rounded-up debt repayment does not fully free the requested collateral. The proportional strategy fails when dust debt remains on a position after proportional distribution but zero collateral margin exists to pass the health check.

The impact is limited to full-exit edge cases and only manifests as tiny discrepancies. Firstly, it means that dust can be left behind. Secondly, callers can work around it by requesting slightly less than the reported totals. Given the restrictions on supported ERC-20 tokens, the practical economic impact is negligible.

Acknowledged:

The 3F has acknowledged the issue, but has decided to keep the code unchanged.

9.3 maxRebalanceLoss Protection Can Be Bypassed via Oracle Price Movement

Informational **Version 4** **Acknowledged**

CS-GRUNT-102

The `PositionManager` enforces a `maxRebalanceLoss` check at the end of every rebalance operation by comparing `totalAssets` before and after the rebalance sequence. This mechanism is designed to limit the damage a compromised `REBALANCER_ROLE` can inflict in a single transaction.

However, the computed value of `totalAssets()` is oracle-dependent: it sums `totalCollateralQuoted()` - `totalBorrowed()` across all borrow modules, where `totalCollateralQuoted()` calls `IOracle(oracle).price()` to convert collateral units into debt-asset units. If the oracle price changes between the computation of `totalAssetsBefore` and `totalAssetsAfter` (for example because of an oracle manipulation or because an oracle update can be sandwiched like this) a favorable price movement can mask an actual loss that exceeds `maxRebalanceLoss`.

A compromised Rebalancer could exploit rising oracle prices by extracting value beyond the configured loss threshold. This requires both a compromised `REBALANCER_ROLE` and an oracle whose price can update in the middle of the rebalancing process.

Acknowledged:

3F has acknowledged the issue, but has decided to keep the code unchanged.

9.4 Balanced Proportional Withdrawals Revert When a Position Exceeds PM LTV

Informational | Version 2 | Acknowledged

CS-GRUNT-100

`_withdrawProportional` with `checkLtv = true` (used by `PositionManagerLP.withdraw()`) distributes debt repayment proportionally to each position's debt share and collateral withdrawal proportionally to each position's collateral share. After repaying debt for a given position, it checks that the collateral to withdraw does not exceed `availableCollateral(ltv)`. If this check fails, it reverts.

If any position in the withdrawal queue has previously drifted slightly above the `PositionManager`'s target LTV (for example due to interest accrual or oracle price movement) then such a proportional withdrawal with balanced inputs, would maintain its health. However, as the health is maintained, it would remain above the `PositionManager`'s target LTV. Hence, the mentioned check would fail and the withdrawal would fail.

As a result, even a perfectly balanced withdrawal (where the overall collateral and debt amounts would maintain the aggregate LTV) can revert because the per-position LTV check fails on the slightly over-leveraged position. This happens even though the withdrawal would maintain the system's overall health. Note that this is effectively equivalent to a proportional burn operation, which would have been allowed in this scenario.

Acknowledged:

3F is aware of this behavior and considers it to be in line with their design decisions:

```
withdraw() accepts arbitrary amounts from the caller,
so the per-position LTV check cannot easily/safely be relaxed.
Callers needing proportional withdrawals should use burn().
```

9.5 Gas Inefficient Implementation of setSupplyQueue

Informational | Version 2 | Acknowledged

CS-GRUNT-088

The function `setSupplyQueue` is less gas efficient when updating the storage array than `setWithdrawalQueue`.

The function `setSupplyQueue` first deletes the original queue (one `SSTORE` to set array's length to zero) and then loops, calling push for each element. At the EVM level, each push operation translates to two distinct storage writes (`SSTORE` for array's length and element). For an array of `n` elements, this results in `2n + 1` storage writes. In contrast, `setWithdrawalQueue` function utilizes a single assignment (`_storage.withdrawalQueue = queue;`), which is optimized by the Solidity compiler. It performs one `SSTORE` to update the array's length and then executes a tight, optimized loop to copy each element from calldata to storage, resulting in `n + 1` storage writes.

The difference is that the direct assignment avoids redundant `SSTORE` operations required to update the array's length in each iteration of the loop.

Acknowledged:

3F has acknowledged the gas inefficiency but has decided to keep the code unchanged for better readability.

9.6 Beacon Proxy Can Use Yul-Optimized Deployment

Informational Version 1 Acknowledged

CS-GRUNT-071

`USCCFundFactory` and `PositionManagerFactory` deploy the high level Solidity implementation of `UpgradeableBeacon` instead of the optimized bytecode version, which is referenced in the documentation. This increases runtime gas cost. However, the Solidity version is verifiable on Etherscan.

Acknowledged:

3F stated:

```
Acknowledged. The current deployment approach is sufficient for our needs.
```

9.7 Descriptor Change Not Paused

Informational Version 1 Acknowledged

CS-GRUNT-044

Generally, it is assumed that pausing, will pause any state-changing function. Every state-changing function except `setDescriptor()` in the `Facility` contract calls `LibStorage.checkNotPaused()` at entry. However, `setDescriptor()` is guarded only by `onlyOwner`.

Hence, if the Facility is paused, the owner can still change the `IIntentDescriptor` contract that generates ERC-6909 token metadata (name, symbol, SVG image). This has no impact on funds, balances, or lifecycle transitions as the descriptor is cosmetic.

Acknowledged:

3F stated:

```
It's okay for the descriptor to be changed without pausing.
The descriptor is a view-only component with no impact on fund operations.
```

9.8 Duplicate Logic in IntentDescriptor

Informational Version 1 Acknowledged

CS-GRUNT-072

The `IntentDescriptor` contract duplicates string concatenation logic across its public functions (`name()`, `symbol()`, `description()`) and their internal counterparts (`_name()`, `_symbol()`, `_description()`), rather than having the public functions delegate to the internal helpers. This

duplication makes the code harder to maintain and increases the risk of inconsistencies if one variant is updated without the other.

Acknowledged:

3F has decided to keep the code unchanged providing the following reasoning:

```
Acknowledged. The duplication is acceptable for readability.
```

9.9 Incorrect Interface Specification of Create in IFund

Informational | Version 1 | Acknowledged

CS-GRUNT-074

The NatSpec documentation in `IFund` for the `create()` function states: "Only callable when state() returns EMPTY.". However, the `create()` function of the `USCCFund` can be called in the `ENDED` state as well.

Acknowledged:

3F has decided to keep the code unchanged.

9.10 Only Partial Events From FacilityFunds

Informational | Version 1 | Acknowledged

CS-GRUNT-051

The Facility contract orchestrates fund lifecycle operations (cancel, commit, unlock, and recover) through `FacilityFunds`, which delegates to an underlying `IFund` implementation like `USCCFund`. While the fund-level contracts emit their own order events, the Facility-level wrapper functions in `FacilityFunds` emit no events for any of these four operations.

The `IFacilityFunds` interface only declares a `CreatingOrder` event for the create flow, leaving cancel, commit, unlock, and recover without corresponding Facility-level event definitions.

Acknowledged:

3F stated:

```
Not needed; in this case the partial events are sufficient and additional events would be redundant.
```

9.11 Unbounded Token Iteration in FacilityLP.claim() Can Cause Gas Exhaustion

Informational | Version 1 | Acknowledged

The `claim()` function in `FacilityLP` iterates over all token entries in an intent's amounts map, performing a `safeTransfer` and storage update for each one. The token set grows each time a new token address is recorded via `commitBalanceSnapshot` or `receiveTokenFrom`. This happens during fund operations, position manager interactions, and swaps.

Theoretically, an intent with a large token set could push `claim()` past the block gas limit, permanently locking LP holders out of their proportional share of all tokens for that intent.

All of those functions are permissioned though, so this would only ever happen if the trusted roles misbehave or are somehow tricked into adding many, many tokens to an intent.

Acknowledged:

3F stated:

```
Acknowledged. The number of tokens in the claim set is bounded by the number of intents
managed by the Facility, which is operationally controlled.
```

10 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.

10.1 Achieved Leverage Will Deviate From Intended Leverage

Note Version 1

The intent flow is asynchronous, with multiple steps executed across different blocks between deposit, bridge loan origination, and final position settlement. During this window, asset prices and Morpho interest rates can shift, altering the effective collateral and debt values relative to what was anticipated.

These price and rate movements change the actual loan-to-value ratio achieved by the time the position is fully constructed. As a result, the realized leverage will rarely if ever match the intended leverage exactly.

This imprecision is inherent to the asynchronous design and cannot be fully eliminated. Users and integrators should expect the final leverage to differ from the target and account for this variance in their risk management.

10.2 Approve Amount Does Not Match Allowance Amount

Note Version 1

For those integrating with the PT/YT contracts it is important to know that:

The `_setAllowance` function in `TokenController` clamps the provided amount to `type(uint128).max` via `FixedPointMathLib.min`. The `allowance` view function then normalizes `type(uint128).max` back to `type(uint256).max`, representing infinite allowance.

As a result, calling `approve` with any value in the range `[2^128, 2^256 - 2]` silently produces an infinite allowance (`2^256 - 1`) that is never consumed on transfers. Integrators expecting a finite, decreasing allowance for values below `type(uint256).max` will observe unexpected behavior, as the reported allowance does not match the value passed to `approve`.

10.3 Facility Proxy Deployment

Note Version 1

The `Facility` is the major contract in the system that is not deployed through a factory. As the `Facility` will be deployed behind a proxy, it is critical that deployment and initialization happen atomically as between the proxy deployment and the `initialize()` call, anyone could call `initialize()` and set themselves as the owner or configure attacker-controlled roles. If this goes undetected, all assets routed through the facility could get stolen.

The `Facility` is deployed as a transparent upgradeable proxy, so it should use the `upgradeAndCall` call in the deployment script to ensure proxy deployment and initialization occur atomically.

10.4 Liquidator Whitelist Requirement Limits Liquidation Participation

Note Version 1

Liquidators who seize collateral from undercollateralized positions receive wUSCC, a wrapped token that must be unwrapped to realize its underlying value. Unwrapping wUSCC into USCC requires the caller to be on the USCC whitelist, imposing a permissioned prerequisite on liquidators.

This means only entities that have completed the USCC whitelisting process can easily participate in liquidations. The restricted access excludes the broader set of MEV searchers and arbitrage bots that typically ensure timely liquidations on permissionless markets. Even if the arbitrageur is whitelisted, the arbitrageur's smart contracts also need to be whitelisted, which adds an additional layer of complexity.

As a result, the effective liquidator pool is significantly smaller than in comparable protocols. Reduced competition among liquidators increases the risk of delayed liquidations, which can lead to bad debt accumulating in the system.

10.5 Mixed Storage Layout From Solady Inheritance Complicates Upgradability

Note Version 1

Central contracts like `Facility` use ERC-7201 Namespaced Storage Layout to isolate storage into deterministic slots, enabling safe upgradability. However, these contracts also inherit from Solady utilities such as `Initializable`, `Multicallable`, and `OwnableRoles`, which use their own fixed storage slots outside the ERC-7201 namespace. Similarly, libraries `LibMintAuth` and `LibTokenController` use their own fixed storage slots.

This creates a mixed storage layout where part of the state follows the namespaced pattern and part relies on Solady's internal slot conventions. If a future upgrade introduces a new Solady parent, the non-namespaced Solady slots could collide with other storage.

Lastly, as many contracts will be deployed as Beacon Proxies, the implementation slot of the Beacon Proxy implementation also needs to be considered in the storage layout.

As a result, the upgrade safety guarantees that ERC-7201 is meant to provide do not fully hold. Developers must manually track both the namespaced and Solady-managed slots to avoid storage collisions during contract upgrades.

10.6 Morpho Supplier Withdrawals Can Leave Intents Unable to Repay Bridge Loans

Note Version 1

The `WrappedAsset` used as collateral in Morpho markets prevents external parties from borrowing against it, giving the protocol predictable control over the borrow side. However, suppliers to the underlying Morpho market retain the ability to withdraw their liquidity at any time without restriction.

If suppliers withdraw enough liquidity, the available debt capacity in the Morpho market shrinks below what an intent needs to borrow. The intent then cannot obtain sufficient debt to repay its bridge loan. This can leave intents in a temporarily stuck state where they are unable to progress through the intent flow.

10.7 NAV Does Not Account for Liquidation Penalty

Note Version 1

The `totalAssets()` function in `PositionManager` computes NAV as quoted collateral minus total debt. However, this calculation does not account for the liquidation penalty that would be incurred if an unhealthy position were liquidated. When a position is liquidatable but has not yet been liquidated, the liquidator receives a bonus proportional to 1 - Health via `preLiquidate()`, meaning the position loses more collateral value than the debt that is repaid. As a result, `totalAssets()` overstates the true realizable value of the fund in case of liquidatable positions. Similarly, standard Morpho liquidations impose a non-zero liquidation penalty.

Since share pricing during deposits and withdrawals relies on `totalAssets()`, this overstatement could lead to mispriced shares: depositors would receive fewer shares than they should, while withdrawers extract more assets.

The facilitator should ensure that no positions are pending liquidation before minting or burning shares to guarantee fair share pricing.

10.8 Oracle Price Jumps

Note Version 1

Oracle prices do not move continuously. They update in discrete jumps whenever a new price is posted on-chain. A sufficiently large price jump can move a position's health from above the pre-liquidation LTV so far that it skips the range where pre-liquidation is preferable over regular Morpho liquidation.

At the time of writing, 3F considers a pre-liquidation LTV of 0.98 and a Morpho market LLTV of 0.98. With such values, a price jump of approximately 1.5% is enough to bypass the pre-liquidation window entirely and expose the position to standard Morpho liquidation and, soon after, bad debt. In such cases, the pre-liquidation mechanism provides no additional protection.

Additionally, the safe LTV should be chosen such that a single realistic oracle price jump cannot make a safe position pre-liquidatable. Otherwise, the risk for users would be too high given the big penalty that they suffer during pre-liquidation.

10.9 OrderId Not Unique When Cancel and Create Are Executed in the Same Block

Note Version 1

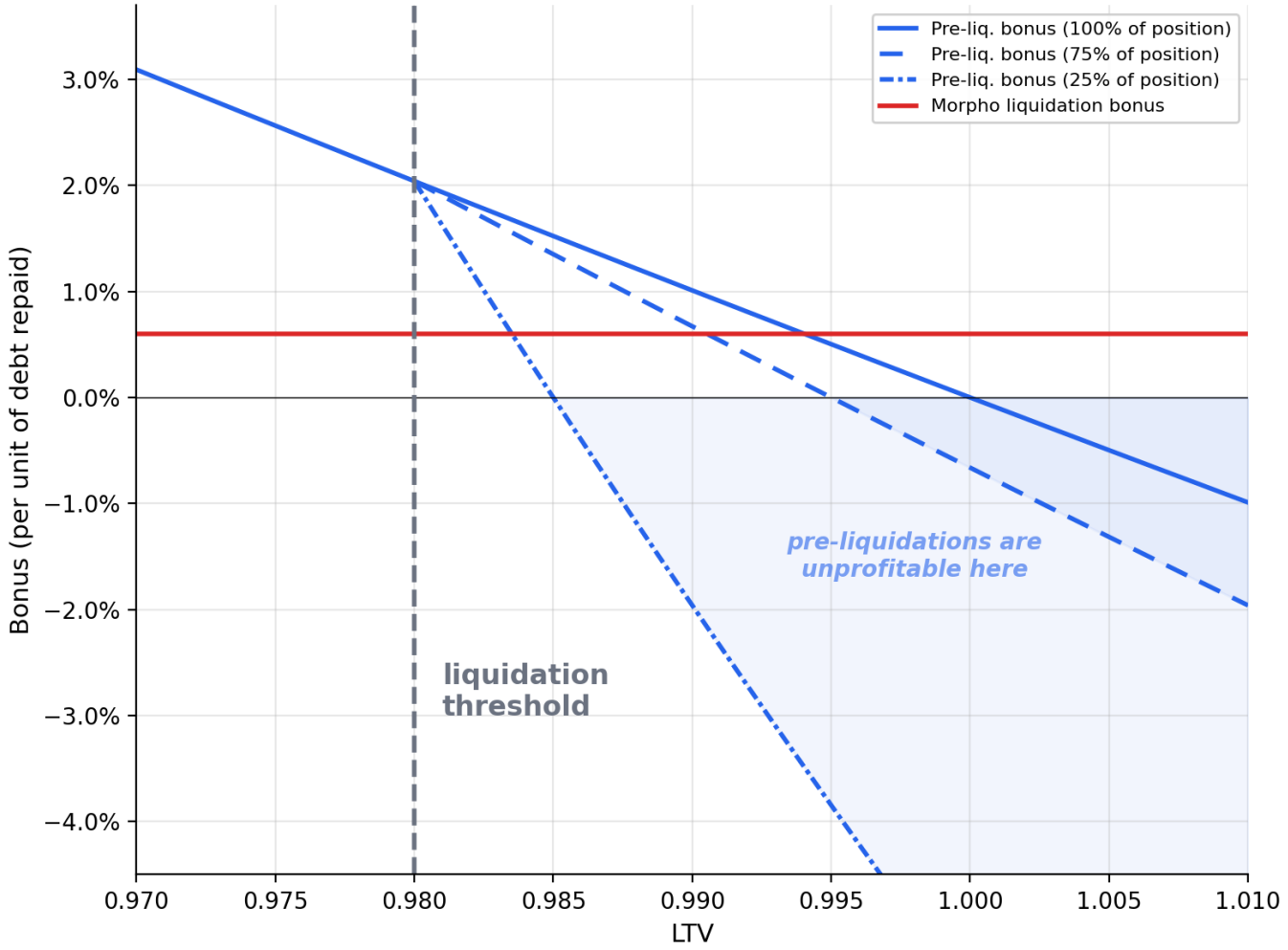
Inside the Facility, the order salt used in the function `FacilityFunds.create()` is `keccak256(abi.encode(address(this), block.timestamp, id))`, and therefore deterministic per block and intent. If a cancel, cancel and re-create with identical parameters occur in the same block, the `orderId` will collide. Since `cancel()` sets the state to `EMPTY` rather than `ENDED`, the subsequent `create()` does not archive the old ID into `endedOrders`, leaving it available for reuse.

The fund handles this correctly internally, but the Facilitator should avoid this pattern as it can confuse external integrators and users tracking orders by ID. Further, the same `orderId` would appear in a `OrderCanceled` and later `OrderUnlocked` events, which should normally not be possible.

10.10 Pre-Liquidation Bonus

Note **Version 1**

The Grunt pre-liquidation mechanism offers a significantly higher liquidation bonus than Morpho Blue's native liquidation for most LLTV configurations. The pre-liquidation bonus scales linearly with the health factor as `1 - LTV`, while Morpho's LIF-based bonus remains constant. As shown below, the pre-liquidation bonus exceeds Morpho's across nearly the entire health range, making pre-liquidations substantially more profitable for liquidators.



Version 4:

When the position's LTV exceeds Morpho's market LLTV, purely proportional partial liquidation would fail Morpho's internal health check, since removing equal fractions of debt and collateral preserves the LTV. The new `_computeRepaidShares` and `_computeSeizedAssets` functions in `MorphoBorrowPosition` adjust the liquidation amounts to account for this: in seized-assets mode, more debt is repaid than proportional; in repaid-shares mode, less collateral is seized, so that the position after the partial liquidation is healthy on Morpho.

This reduces the liquidator's bonus below the pure `1 - LTV` line once LTV exceeds the LLTV threshold. As visible in the graph, partial pre-liquidations seizing 25% or 75% of the position become unprofitable (negative bonus) earlier than a full liquidation, since the remaining position must be brought back to the LLTV. Note that the graph plots the case where pre-liquidation LTV equals the market's LLTV, which might not be the case.

10.11 Privileged Roles Can Adjust Leverage via Bridge Loan Size

Note Version 1

When a Position Manager is the target asset of an intent, privileged roles can influence the effective leverage of that Position Manager by adjusting the size of the bridge loan used during the intent flow. A larger bridge loan increases leverage while a smaller one decreases it.

During the `deposit` into the Position Manager, the intent will receive Position Manager shares according to the depositors' net position. The bridge loan carries a fee that is deducted from the depositors' net position. Different loan sizes produce different fee amounts, and hence different net positions.

Depositors have no control over this decision. But the bridge loan size chosen by the privileged roles determines the amount of Position Manager shares they receive. Therefore, privileged roles have to use this power carefully as adjusting the leverage can either benefit the depositors or the already existing Position Manager shareholders.

10.12 Security Considerations When Updating the Intent Target

Note Version 1

`FacilityIntents.updateTarget()` allows the owner to overwrite the target asset and guard key of an intent. This action is permitted in any intent state (depositing, resolving, or resolved). The following should be considered by the owner when calling this function:

1. Always: Users and integrating contracts must be aware that they might receive a new target asset. They may expect DAI but receive USDC.
2. If both the deposit asset and the new target asset are position managers: Both position managers must have the same collateral and debt as enforced by `_checkAssetsAndGuardKey`.
3. If the deposit asset is not a PM but the new target asset is a PM: The new target becomes the guard key. Existing funds and requests were validated against the previous guard key's assets at the time `setFund` or `setRequest` was called; `updateTarget` does not re-run those checks. If the new guard key PM has different collateral or debt assets, the existing funds and request could become incompatible with the intent.

10.13 Security Considerations for DeFi Integrators

Note Version 1

10.13.1 YT

The `VaultController` implements a dual-token system where Principal Tokens (PT) are senior and Yield Tokens (YT) are subordinated. The core asset distribution formula (`pAssets = min(totalAssets, ptSupply)` with `yAssets = totalAssets - pAssets`) creates non-standard ERC-4626 behavior.

Standard ERC-4626 view functions such as `convertToShares()`, `previewDeposit()`, `previewWithdraw()`, and `convertToAssets()` behave in specific and potentially unexpected ways. `previewDeposit()` always returns zero because direct deposits are disabled. `convertToAssets()`

for YT shares returns zero whenever the vault's total assets are at or below the PT supply (the underfunded regime), then jumps discontinuously to a positive value once assets exceed the PT supply. When YT supply is zero but yield assets exist, `convertToShares()` returns `type(uint256).max`. These edge cases mean that protocols relying on smooth, continuous share pricing (such as lending markets, AMMs, or yield aggregators) may encounter unexpected behavior.

Given the subordinated nature of YT, the disabled deposit path, and the discontinuous pricing behavior, YT tokens are generally not recommended for integration into DeFi protocols.

10.13.2 Intent Shares

Pricing Intent Shares is very complex due to the dynamic nature of bridge loans, `PositionManager` shares and the asynchronous settlements. Hence, it is not recommended to integrate Intent Shares in a way that requires pricing them.

10.13.3 PositionManager Shares

When pricing PositionManager shares, integrators should be aware that the PM's `totalAssets()` can be inflated by third-party donations directly to Morpho Blue. Furthermore, integrators need to consider `pendingFees()` when calculating the value of a PM share as these fees are owed and additional shares will be minted. Hence, current shares will be diluted by future fee settlements.

10.14 Security Considerations for Guardian Signers

Note Version 1

1. **General.** Considerations for all guardian-gated actions
   - The signer address is not hashed into any EIP-712 digest, so a signature is not bound to a specific SC wallet. If the same EOA is an authorized signer on multiple SC wallets, a signature produced for one wallet can be replayed on another whenever the wallet's `isValidSignature()` function verifies against the raw application digest without first deriving a wallet-specific digest. Some older SW contracts did this; modern wallets such as Safe compute a wallet-specific digest. Signers using SC wallets must ensure their implementation includes the wallet address in the validation digest.
   - Set tight deadlines: Signed digests cannot be nullified, so the deadline is the only expiry mechanism.
2. **Swaps.** Considerations for `swap()`;
   - Verify no malicious token addresses in the swap parameters.
   - Check that pricing is fair (see: Swap Pricing Risks)
3. **setFund.** Considerations for `setFund()`.
   - Fund was deployed by the correctly initialized, respective Fund Factory.
   - Fully trusted roles (including owner, upgrade admin, admin, operator) are multi-signature wallets. Their signers are controlled by trusted entities of this project.
   - Semi-trusted roles have been assigned to correct entities and are reasonably secured.
   - Facility has the necessary permissions.
   - Correct and trustworthy oracles have been selected.

- Fund-specific configuration is correct.
- No unnecessary roles/permissions have been granted.

4. **setRequest.** Considerations for `setRequest()`.
   - Request deployed by the canonical `RequestFactory` (which has been set up correctly with proper roles).
   - Fully trusted roles (including owner, upgrade admins, admin) are multi-signature wallets. Their signers are controlled by trusted entities of this project.
   - Semi-trusted roles have been assigned to correct entities and are reasonably secured.
   - Facility has the necessary permissions.
   - `repaymentDeadline` and `mintToRepaidDelay` give sufficient time to detect a compromised facilitator or compromised consumer.
   - Fund-specific configuration (including name and symbol) is correct.
   - No unnecessary roles/permissions have been granted.

10.15 Security Considerations for Liquidators

Note Version 2

Liquidators should consider the following properties of pre-liquidations before integrating them:

- **Frontrunning:** Another caller can liquidate the same position first, reducing its debt or collateral. If the liquidator then attempts to seize more collateral or repay more debt than the position holds, `preLiquidate()` reverts.
- **Rounding against the liquidator:** Debt repayment rounds up (`mulDivUp`) and collateral seizure rounds down (`mulDiv`), reducing the liquidator's profit.
- **Capped liquidation bonus:** The pre-liquidation mechanism distributes collateral proportionally, yielding a bonus of `1 - LTV` as a fraction of the seized collateral value. However, when the position's LTV exceeds the Morpho market LLTV, a purely proportional *partial* liquidation would fail Morpho's internal health check, since removing equal fractions of debt and collateral preserves the LTV. To pass the health check, the mechanism adjusts: in `seizedAssets` mode the liquidator must repay *more* debt than the proportional amount, and in `repaidShares` mode the liquidator receives *less* collateral than the proportional amount. This can make **partial** liquidations **unprofitable**.

Liquidators should operate through a smart contract that reads fresh position and market data to compute fresh data for `seizedAssets` or `repaidShares`. Further, their contract should enforce that their liquidations are profitable, e.g., by making use of the `onPreLiquidate` callback.

10.16 Security Considerations for the Facilitator

Note Version 1

Generally, when using functions like `swap`, `setFund` and `setRequest`, the Facilitator should perform the same checks that Guardians perform. See Security Considerations for Guardian Signers for details. However, this is extremely important if the intent's `quorum` is set to 0.

When creating an order inside a fund using `create`, the Facilitator should consider asynchronous settlement and avoid setting a `minAmountOut` which is too high as that would likely delay the overall fund settlement.

The Facilitator should consider the following behavior when interacting with the `PositionManager`.

- The Facilitator must not deposit or withdraw into the `PositionManager` whenever the PM is currently not at target leverage. Leveraging up or down can result in shares minted or burned that is not in line with the action itself.
- The Facilitator should not deposit, withdraw or burn whenever there is a module with bad debt. See Behavior of PM Deposits, Withdrawals and Burns with Bad Debt for an explanation.
- They should not deposit, withdraw or burn when there are liquidatable positions, see NAV Does Not Account for Liquidation Penalty.

10.17 Share Price Deflation After Full Pre-Liquidation

Note **Version 1**

If a `PositionManager` holds a single borrow position that gets fully pre-liquidated, `totalAssets()` drops to zero while the existing share `totalSupply` remains unchanged. The outstanding shares effectively become worthless but still count toward the denominator in the share-minting formula.

On the next deposit, `convertToShares` computes:

```
assets * (totalSupply + VIRTUAL_SHARES) / (0 + VIRTUAL_ASSETS)
```

Because `totalSupply` can be orders of magnitude larger than `VIRTUAL_ASSETS` (which is 1), the minted share amount can be extremely large. This inflated supply could cause arithmetic overflows in subsequent operations that multiply shares by asset values, such as `burn`'s proportional calculations or fee accrual logic.

10.18 Share Price Inflation Possible Despite Mitigation

Note **Version 1**

The `PositionManager` uses a virtual shares offset:

- `VIRTUAL_SHARES = 1e6`
- `VIRTUAL_ASSETS = 1`

This makes share price inflation economically unprofitable (under certain assumptions) for an attacker. However, the inflation itself remains possible. An attacker can still donate directly via Morpho's `repay` to inflate `totalAssets` without minting shares.

While existing shareholders are not at risk of loss due to the virtual offset, external integrations that price `PositionManager` shares or use them as collateral must account for the possibility of artificially inflated share prices and sudden price increases.

10.18.1 Updates in Version 4

Virtual shares offset is now computed based on the decimals of the underlying debt asset, as:

- `VIRTUAL_SHARES = 10 ** (18 - decimals)`

Hence, for a debt asset with 6 decimals, the virtual shares offset would be `10 ** (18 - 6) = 10 ** 12`, while for a debt asset with 18 decimals, the virtual shares offset would be `10 ** (18 - 18) = 1`.

As a consequence, `PositionManager` shares always have 18 decimals, regardless of the decimals of the underlying debt asset. As discussed in Share Inflation via Direct Morpho Collateral Donation, profitable inflation attacks are realistic for debt assets with 18 decimals, but not for debt assets with 6 decimals, such as USDC.

10.19 Supply Queue Deposits Can Be Manipulated by Dominant Lenders

Note Version 1

The `PositionManager`'s supply queue determines which borrow positions receive collateral and take on debt during deposits. If a Morpho market referenced by a queue entry has a single party supplying a significant portion of the loan token liquidity, that party can influence how deposits are processed.

By front-running a deposit transaction with a withdrawal of their supplied liquidity, the dominant lender can reduce the available liquidity in targeted markets. This forces the deposit to skip those queue entries or revert entirely if insufficient liquidity remains across all entries. The lender can then re-supply after the deposit, temporarily causing higher interest rates for existing borrowers in the affected markets and selectively disrupting the intended deposit distribution.

10.20 Swap Pricing Risks

Note Version 1

The `swap()` function in `FacilitySwap` transfers fixed token amounts between two intents with no on-chain price oracle validation. Swap amounts are determined off-chain and authorized via EIP-712 signatures by guardians holding the global `GUARDIAN_ROLE`.

Guardians should consider the following when approving swaps:

- Validating swap prices with up to date oracle prices.
- Accounting for any pending liquidations or bad debt of PM positions.
- Setting tight deadlines to prevent execution at stale prices.