GA GUARDIAN

Tenor

Morpho Integration

Security Assessment

June 1st, 2026

Summary

Audit Firm Guardian

Prepared By Robert Reigada, Wafflemakr, Nicholas Chew

Client Firm Tenor

Final Report Date June 1st, 2026

Audit Summary

Tenor engaged Guardian to perform a review of its Morpho integration contracts. From April 20 through May 4, 2026, Guardian reviewed the code in scope and recorded the vulnerabilities found in the following report.

Confidence Ranking

Given the lack of High and Critical issues detected during the main review, Guardian assigns a Confidence Ranking of 4 to the protocol. Guardian advises the protocol to address issues thoroughly and consider a targeted follow-up audit depending on code changes. For detailed understanding of the Guardian Confidence Ranking, please see the rubric on the following page.

Guardian Confidence Ranking

Confidence Ranking	Definition and Recommendation	Risk Profile
5: Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. **Recommendation:** Code is highly secure at time of audit. Low risk of latent critical issues.	0 High/Critical findings and few Low/Medium severity findings.
4: High Confidence	Code is clean, well-structured, and adheres to best practices. Only 1 Significant issue was uncovered per week. Design patterns are sound, and test coverage is strong. **Recommendation:** Suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity issues. Varied Low severity findings.
3: Moderate Confidence	Medium-severity and occasional High-severity issues found. Code is functional, but there are concerning areas (e.g., weak modularity, risky patterns). No critical design flaws, though some patterns could lead to issues in edge cases. **Recommendation:** Address issues thoroughly and consider a targeted follow-up audit depending on code changes.	1-2 High/Critical findings per engagement week.
2: Low Confidence	Code shows frequent emergence of Critical/High vulnerabilities. Audit revealed recurring anti-patterns, weak test coverage, or unclear logic. These characteristics suggest a high likelihood of latent issues. **Recommendation:** Post-audit development and a second audit cycle are strongly advised.	2-4 High/Critical findings per engagement week. Or additional High/Critical findings uncovered in remediation review which have not been resolved and confirmed by Guardian.
1: Very Low Confidence	Code has systemic issues. Multiple High/Critical findings (≥5/week), poor security posture, and design flaws that introduce compounding risks. Safety cannot be assured. **Recommendation:** Halt deployment and seek a comprehensive re-audit after substantial refactoring.	≥5 High/Critical findings and overall systemic flaws.

Table of Contents

Project Overview

Project Summary

Project Name	Tenor
Codebase	https://github.com/Shippooor-Labs/tenor-morpho-v2-contracts-2
Commit(s)	**Main Review:** 75ee03e8502b955522445acb8142047c3e0ad652 **Remediation Review:** 13b94b0acd7597f3fffc6ac9ebdfb2517e960480

Audit Summary

Delivery Date	June 1, 2026
Audit Methodology	Static Analysis, Manual Review

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
Critical	0	0	0	0	0	0
High	1	0	0	0	0	1
Medium	25	0	0	17	0	8
Low	22	0	0	16	0	6
Info	29	0	0	18	0	11

Audit Scope & Methodology

RenewalIntentRatifier.sol
RenewalIntentRegistry.sol
RenewalIntentTakeOnBehalf.sol
MidnightAuthorizationAdapter.sol
RenewalIntentTakeOnBehalfAdapterBase.sol
TakeRouterAdapterBase.sol
TenorAdapter.sol
MorphoV1ToV2BorrowCallback.sol
MorphoV1ToV2LendCallback.sol
MorphoV2SupplyCollateralCallback.sol
MorphoV2SupplyVaultSharesCallback.sol
MorphoV2ToV1BorrowCallback.sol
MorphoV2ToV1LendCallback.sol
MorphoV2ToV2BorrowRepay.sol
MorphoV2ToV2LendWithdrawable.sol
MorphoV2WithdrawVaultSharesCallback.sol
DelayedLiquidationGateFactory.sol
MidnightAllowlistGateFactory.sol
OracleWithValidationCheckFactory.sol
VaultV2AllowlistGateFactory.sol
DelayedLiquidationGate.sol
MidnightAllowlistGate.sol
PauseTakeGate.sol
VaultV2AllowlistGate.sol
CallbackLib.sol
CollateralTransferLib.sol
Constants.sol
KeyLib.sol
PriceLib.sol
RatePointLib.sol
ToIdLib.sol
OracleWithValidationCheck.sol
MidnightVaultExecutor.sol
TakeRouter.sol
MidnightLib.sol
RouterLib.sol
ClampLib.sol
SupplyCollateralCallbackClamp.sol
V1ToV2LendClamp.sol
V2ToV2BorrowRepayClamp.sol
V2ToV2LendWithdrawableClamp.sol
VaultSupplyClamp.sol
VaultWithdrawClamp.sol
FourWeekCadence.sol
PausableStaticRatePolicy.sol
StaticRatePolicy.sol

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: *High*	Impact: *Medium*	Impact: *Low*
Likelihood: *High*	🔴 Critical	🟠 High	🟡 Medium
Likelihood: *Medium*	🟠 High	🟡 Medium	🟢 Low
Likelihood: *Low*	🟡 Medium	🟢 Low	🟢 Low

Impact

High Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.

Medium A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.

Low Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

High The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.

Medium An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.

Low Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts. Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	Overdue V2-V2 Borrow Renewals Bypass Max Rate	Logical Error	High	Resolved
M-01	Executor Lets Callers Sweep Shared Balance	Unexpected Behavior	Medium	Acknowledged
M-02	Primary-based Deviation Can Erase Buffer	Logical Error	Medium	Acknowledged
M-03	V2-to-V2 Lend Can Buy Bad Credit	Unexpected Behavior	Medium	Acknowledged
M-04	Callback Spends Shared Loan Balance	Unexpected Behavior	Medium	Acknowledged
M-05	Ratifiers Miss Seller Receiver	Unexpected Behavior	Medium	Acknowledged
M-06	TakeRouter Breaks Caller-sensitive Policies	Logical Error	Medium	Resolved
M-07	onFlashLoan Missing Approval Breaks Bundled Ops	Logical Error	Medium	Resolved
M-08	Incorrect Math Inversion For SELL + Fee	Rounding	Medium	Resolved
M-09	Executor Hides Real Liquidator From Gate	Validation	Medium	Resolved
M-10	Zero Oracle Price Gives Free Collateral	Unexpected Behavior	Medium	Acknowledged
M-11	Keeper Params Do Not Guarantee Profitability	Logical Error	Medium	Acknowledged
M-12	Temporary Grants Bypass Router Limits	Validation	Medium	Acknowledged

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
M-13	Flash Callback Spends Pending Reentry	Validation	Medium	Acknowledged
M-14	Source Unlocked During Borrow Renewal	Reentrancy	Medium	Acknowledged
M-15	executeAndConsume Burns Keeper consumeGroup	Logical Error	Medium	Resolved
M-16	Keeper Adapter Callbacks Reject Delegation	Logical Error	Medium	Resolved
M-17	Adapter Callbacks Cannot Batch Reentries	Compatibility	Medium	Resolved
M-18	executeAndConsume Accepts Intervening Fills	Unexpected Behavior	Medium	Acknowledged
M-19	Clamps Ignore Revoked Callback Auth / Allowances	DoS	Medium	Acknowledged
M-20	Tenor Prioritizes Midnight's Trading Fees	Logical Error	Medium	Acknowledged
M-21	Keeper Gets Full Midnight Control	Access Control	Medium	Acknowledged
M-22	TenorAdapter Bypasses Midnight's Repay Auth	Validation	Medium	Resolved
M-23	Arbitrary Policies Can Gas Grief Keepers	DoS	Medium	Acknowledged
M-24	Permissionless First Touch Freezes Default Fees	Unexpected Behavior	Medium	Acknowledged
M-25	Fresh Credit Drains Old Repayments	Validation	Medium	Acknowledged

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
L-01	allowRevert Misses Pre-dispatch Reverts	Unexpected Behavior	Low	Acknowledged
L-02	Frozen Gate Trusts Fee Recipients	Trust Assumptions	Low	Acknowledged
L-03	Vault-share Exit Ignores Vault Withdrawal Limits	Validation	Low	Acknowledged
L-04	Zero-price Asset Caps Act Unbounded	Unexpected Behavior	Low	Acknowledged
L-05	isFinalFill Never True With Non-Zero Fee	Rounding	Low	Acknowledged
L-06	Fee Recipients Can Get Stuck Shares	Unexpected Behavior	Low	Resolved
L-07	Clamps Ignore Active Collateral Cap	Validation	Low	Resolved
L-08	Fragmented Fills Skip Discrete Collateral	Rounding	Low	Acknowledged
L-09	Target-side Netting Changes Renewal Semantics	Unexpected Behavior	Low	Acknowledged
L-10	Oracle Validation Can Fail Open Or Freeze	Oracle	Low	Acknowledged
L-11	Vault Executor Redeems Without Asset Floors	Validation	Low	Resolved
L-12	Fee Adjuster Overshoots maxFill	Rounding	Low	Resolved
L-13	Griefing V1=>V2 Migration	Gas Griefing	Low	Acknowledged

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
L-14	Router Fee Metadata Under-reports Fees	Unexpected Behavior	Low	Acknowledged
L-15	Bidirectional Intents Enable Fee-draining Churn	Gaming	Low	Acknowledged
I-01	V1->V2 Borrow Renewal Can Spend Target Credit	Validation	Info	Acknowledged
I-02	Stale Grace Timer Survives Healthy Recovery	Unexpected Behavior	Info	Acknowledged
I-03	V2-to-V1 Fees Bypass Rate Limits	Validation	Info	Acknowledged
I-04	Sell Takes Can Spend Router-held Tokens	Warning	Info	Acknowledged
I-05	Supply Collateral Clamp Quotes Unhealthy Fills	Warning	Info	Acknowledged
I-06	TENOR_VAULT_V2 Clamp Reads Wrong Data	Unexpected Behavior	Info	Resolved
I-07	Vault Supply Clamp Uses An Unbound Tick	Unexpected Behavior	Info	Resolved
I-08	Vault Supply Clamp Omits Solvency Bound	Validation	Info	Resolved
I-09	Grace Period Skipped Once Obligation Matures	Validation	Info	Resolved
I-10	Factory Allows Near-total Oracle Deviation	Validation	Info	Acknowledged
I-11	Zero-owner Oracle Docs Cannot Deploy	Documentation	Info	Resolved

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
I-12	Lender Rate Floors Bypassed After Early Repay	Logical Error	Info	Acknowledged
I-13	Withdrawals Escape Lazy Bad Debt	Validation	Info	Acknowledged
I-14	Renewal Skips Collateral Missing From Target	Unexpected Behavior	Info	Resolved
I-15	TakeRouter Returns Adjusted Totals Only	Unexpected Behavior	Info	Acknowledged
I-16	V2-to-V2 Lend Callback Self-funds Fresh Credit	Unexpected Behavior	Info	Acknowledged
I-17	Renewal Grants Survive Midnight Revocation	Trust Assumptions	Info	Acknowledged
I-18	V2-to-V1 Borrow Exits Lack Target Health Buffer	Validation	Info	Acknowledged
I-19	Stale Documentation	Documentation	Info	Resolved
I-20	Sentinel Silently No-ops Takes	Logical Error	Info	Resolved
I-21	Stable Max Sentinel Nets Gas	Unexpected Behavior	Info	Resolved
I-22	Delayed Gate Blocks Gated Vault Liquidations	DoS	Info	Acknowledged

H-01 | Overdue V2-V2 Borrow Renewals Bypass Max Rate

Category	Severity	Location	Status
Logical Error	High	BaseRenewalRatifier.sol	Resolved

Description

`BaseRenewalRatifier` intentionally allows V2-source renewals after the source obligation has matured. `BaseRenewalRatifier._ratifyWindow()` only rejects executions that are too early, so once the clock passes `sourceMaturity` the renewal remains permanently executable. That behavior is documented and tested. The problem is that the downstream rate check does not switch to a post-maturity duration model when that overdue state is reached.

For V2-to-V2 renewals, `BaseRenewalRatifier._computeDuration()` always returns `targetMaturity - sourceMaturity`. That is correct only while the source obligation is still alive. After maturity, the new debt does not begin at the expired `sourceMaturity`; it begins when the keeper executes the renewal. The economically correct duration for the borrow-side rate check is therefore `targetMaturity - block.timestamp`, or equivalently `targetMaturity - max(block.timestamp, sourceMaturity)`. That distinction matters because `PriceLib.computePrice()` is inversely proportional to duration. A longer duration produces a lower accepted price. On the borrower path, `PriceLib.satisfiesRateLimit()` treats a lower accepted price as a looser rate ceiling because the borrower only needs `(assets - fee) * WAD` to stay above `units * price`. In other words, once the ratifier uses an overdue-inclusive duration, it accepts prices that imply a materially higher realized rate over the actual remaining term of the renewed debt.

The effect scales directly with overdue time. If a borrower set `limitRatePerSecond` to cap a renewal at rate `r`, the post-maturity V2-to-V2 check accepts the price for duration `T + D`, where `D` is time overdue and `T` is the actual time from execution until the target maturity. The debt that is actually created lasts only `T`, so the same accepted discount corresponds to an effective ceiling of `r * (T + D) / T`. A renewal that is overdue by sixty days and rolls into a new thirty-day target therefore weakens a ten percent ceiling into roughly a thirty percent realized ceiling.

The ratifier explicitly accepts the overdue state, then computes borrower protection from the wrong time origin. The issue is reachable on live accounting because the source debt is already matured, the target maturity must still satisfy the user's `minDuration` and `maxDuration` relative to the current timestamp, and the callback opens fresh target debt immediately after ratification.

The lender side does not have the same risk direction. Reusing `targetMaturity - sourceMaturity` on a lend renewal makes the floor stricter, not looser. The unsafe manifestation is specifically the V2-to-V2 borrow path because borrower ceilings are enforced with the seller-side inequality.

Recommendation

If overdue rolling is not intended, close the V2 renewal window at `sourceMaturity` and revert once the source obligation is overdue. That is the simplest fix because it restores the original interpretation of `sourceMaturity` as the start of the renewed term. If overdue rolling is intended, keep the window semantics explicit but compute duration from the actual renewal start time:

```
    function _computeDuration(address callback, uint256 sourceMaturity, uint256 targetMaturity)
    internal
    view
    ...
    ...
    }
```

Resolution

Tenor Team: The issue was resolved in PR#427.

M-01 | Executor Lets Callers Sweep Shared Balance

Category	Severity	Location	Status
Unexpected Behavior	Medium	MidnightVaultExecutor.sol	Acknowledged

Description

`repayAndWithdrawCollateral` and `liquidateAndRedeem` both treat the executor's entire `loanToken` balance as if it belonged to the current caller. In `repayAndWithdrawCollateral`, the function calls `Midnight.repay`, then transfers `IERC20(obligation.loanToken).balanceOf(address(this))` to `onBehalf`. It never tracks how much of that balance came from the current repayment. The function also accepts `repayUnits == 0` and `sharesToWithdraw == 0`, so any account can call it with `onBehalf = msg.sender` and withdraw every loan token already sitting on the executor.

`liquidateAndRedeem` has the same accounting problem from the funding side. It grants Midnight a max allowance and lets `Midnight.liquidate` pull `repaidUnits` from whatever balance the executor already holds. The integration tests model an EOA liquidator by pre-funding the executor before calling `liquidateAndRedeem` (`test/integration/MidnightVaultExecutorIntegration.t.sol:823-832`). When Midnight computes `actualRepaidUnits` below the prefunded amount, the helper never refunds the surplus. That residue stays on the public executor and becomes claimable by the next caller through `repayAndWithdrawCollateral`, or usable as free repayment capital in a later liquidation. This directly contradicts the documented `EXEC-3` invariant in `PROPERTIES.md:316`.

Consequently any loan tokens stranded on the executor are not isolated to the account that supplied them. They are globally spendable. This is a direct loss of funds for liquidators or users who overfund the executor and the contract already acknowledges one source of residual dust in the mint path. The bug does not directly drain the normal protocol fee sinks: Tenor callback fees are sent to the configured `feeRecipient` and Midnight trading fees accrue inside Midnight as `claimableTradingFee`. The practical theft target is loan-token residue that actually lands on the executor, including unused liquidation prefunds, accidental transfers and executor-held dust.

Recommendation

Stop using the executor's aggregate token balance as the payment and refund source. Snapshot the loan-token balance at function entry and only refund the positive delta created by the current call. Reject `repayAndWithdrawCollateral` when both `repayUnits` and `sharesToWithdraw` are zero. For `liquidateAndRedeem`, either require exact callback-based funding, or refund `postBalance - preBalance` to `msg.sender` and revert if the executor starts with a nonzero loan-token balance. If unexpected balances must remain recoverable, add a dedicated privileged rescue mechanism instead of a public sweep.

Resolution

Tenor Team: The issue was resolved in PR#449.

M-02 | Primary-based Deviation Can Erase Buffer

Category	Severity	Location	Status
Logical Error	Medium	OracleWithValidationCheck.sol	Acknowledged

Description

`price()` scales `MAX_ORACLE_DEVIATION` by `primaryPrice`, then returns that same `primaryPrice` if the check passes. When the primary oracle is the high side, this accepts `primaryPrice <= validationPrice / (1 - d)` rather than `primaryPrice <= validationPrice * (1 + d)`. Consequently a configured 5% threshold actually allows about 5.263% overpricing against the validation oracle.

This matters because the README recommends sizing `MAX_ORACLE_DEVIATION` below `(1 - Liquidation LTV)` for lending markets. With a 95% LLTV market and a 5% threshold, this wrapper can still return a price high enough to let the protocol lend 100% of the validation-based collateral value. Even at 94.5% LLTV, the remaining buffer is only about 0.53%. A borrower who can push the primary oracle up within this accepted band can open positions that are effectively at or near full LTV on the validation price, so negligible market movement or routine interest accrual can leave bad debt.

Recommendation

Measure the allowed deviation from `validationPrice`, or more conservatively from `min(primaryPrice, validationPrice)`, instead of `primaryPrice`. If the current formula must remain, do not size `MAX_ORACLE_DEVIATION` directly from `(1 - LLTV)`. Document and enforce the stricter bound `MAX_ORACLE_DEVIATION <= (1 - LLTV) / (2 - LLTV)` for markets that rely on this wrapper to preserve liquidation margin.

Resolution

Tenor Team: The issue was resolved in commit 14660ca.

M-03 | V2-to-V2 Lend Can Buy Bad Credit

Category	Severity	Location	Status
Unexpected Behavior	Medium	MorphoV2ToV2LendWithdrawable.sol	Acknowledged

Description

`MorphoV2ToV2LendWithdrawable.onBuy` only checks that the source and target obligations use the same loan token before withdrawing the lender's source credit. It does not prove that the target fill created fresh borrower debt backed by currently healthy collateral. That matters when the seller already has credit on the target obligation. `Midnight.take()` updates positions before the callback runs. For a sell offer, it first decreases the seller's existing credit and only creates seller debt for any excess units:

```
    uint256 sellerCreditDecrease = UtilsLib.min(units, sellerPos.credit);
    uint256 sellerDebtIncrease = units - sellerCreditDecrease;
```

If `sellerDebtIncrease == 0`, the seller exits an existing lending position instead of opening a new borrow. Midnight's final `isLiquidatable` check then returns immediately because the seller has no debt, so no collateral oracle is queried. Consequently, a lender's renewal can spend withdrawable source credit to buy target credit that is already backed by stale, worthless, or otherwise bad collateral. This can happen after the lender has configured the target market and authorized the callback, because any keeper can execute a matching renewal while the callback only enforces loan-token equality and source-credit availability.

Recommendation

Require explicit opt-in before a V2-to-V2 lend renewal buys existing target credit. The safer default is to reject fills where the target seller has enough credit to avoid a debt increase, or to require the router or ratifier to verify a fresh target health/oracle check for the underlying target obligation before the source withdrawal is allowed. If buying existing target credit is intended, expose it as a separate operation with distinct user consent and event semantics. Do not treat it as the same renewal as creating new target lending exposure against fresh borrower debt.

Resolution

Tenor Team: Acknowledged.

M-04 | Callback Spends Shared Loan Balance

Category	Severity	Location	Status
Unexpected Behavior	Medium	MorphoV2SupplyVaultSharesCallback.sol	Acknowledged

Description

`onSell` assumes the callback's current `loanToken` balance came from the current Midnight fill. The function never checks that `receiverIfMakerIsSeller` sent `sellerAssets` to the callback, and it never snapshots the callback's balance before using `totalDeposit`. If the callback already holds `loanToken`, a seller can sign a sell offer that uses this callback but routes `receiverIfMakerIsSeller` to another address. Midnight will send the borrowed `sellerAssets` to that receiver, then `onSell` will pull only `amountFromSeller` from the seller. The later `deposit(totalDeposit, address(this))` can consume the pre-existing callback balance for the missing `sellerAssets`, mint vault shares, and supply those shares as the seller's collateral.

Consequently any loan tokens accidentally sent to this public callback, or otherwise stranded there, are not isolated. A later seller can convert that shared balance into their own vault-share collateral and unwind the position to recover the value.

Recommendation

Do not use the callback's aggregate token balance as implicit funding. At minimum, snapshot `IERC20(loanToken).balanceOf(address(this))` at entry and require that the balance equals `sellerAssets` before any seller top-up is pulled. After pulling `amountFromSeller`, require that the balance equals `totalDeposit`. This makes wrong receiver configuration and pre-existing balances fail before the vault deposit. If stranded balances need recovery, add an explicit rescue function instead of letting future fills consume them.

Resolution

Tenor Team: The issue was resolved in PR#451.

M-05 | Ratifiers Miss Seller Receiver

Category	Severity	Location	Status
Unexpected Behavior	Medium	RenewalIntentTakeOnBehalf.sol	Acknowledged

Description

`RenewalIntentTakeOnBehalf.take` forwards a caller-supplied `receiver` into `MORPHO_MIDNIGHT.take`, but it does not pass that receiver to the user-side ratifier. The ratifier only sees `caller`, `takerCallback`, `intent`, `offer`, `takerCallbackData` and `ratifierData`:

```
    IIntentRatifier(intent.ratifier)
    .onIntentRatify(msg.sender, takerCallback, intent, offerData.offer, takerCallbackData,
ratifierData);
    MORPHO_MIDNIGHT.take(
    ...
    offerData.proof
    );
```

This is a dangerous blind spot for renewal routes where `intent.user` is the seller side of the Midnight trade. In those routes, Midnight sends `sellerAssets` to `receiverIfTakerIsSeller` before executing the seller callback. The in-repo seller-side renewal callbacks explicitly rely on that receiver being the callback contract so the callback can spend the fresh fill proceeds. For example, `MorphoV2ToV2BorrowRepay` states that the offer receiver must be the callback, then transfers `fee`, approves `repayBudget`, repays the source obligation and moves collateral without proving that the current fill actually funded the callback.

An arbitrary keeper can choose a different `receiver` that the ratifier cannot inspect. If the callback already holds enough loan-token balance, the fill can still complete: the keeper diverts the current `sellerAssets` to the chosen receiver, while the callback spends its old balance to pay the fee and repay or deposit on behalf of the user. The old balance can be accidental token residue, or it can be created by the callbacks themselves when `feeRecipient == address(this)`, because the fee transfer becomes a no-op and only `sellerAssets - fee` is spent afterward. The same aggregate-balance pattern appears in `src/callbacks/MorphoV1ToV2BorrowCallback.sol` and `src/callbacks/MorphoV2ToV1LendCallback.sol`.

The root cause is that the user-side ratifier is supposed to police renewal shape, but one of the most important value-routing fields is omitted from the ratification payload.

Recommendation

Include `receiver` in `IIntentRatifier.onIntentRatify` and require the relevant ratifiers to validate it for each renewal route. Seller-side renewal routes should reject any receiver other than the expected callback contract.

The callbacks should also enforce fresh funding instead of relying on their aggregate token balance. Consider asserting that the callback's loan-token balance increased by exactly the expected current-fill amount before spending, and that no unexpected loan-token balance remains afterward. Also reject `feeRecipient == address(this)` or add an explicit controlled sweep path for accidental balances.

Resolution

Tenor Team: Acknowledged.

M-06 | TakeRouter Breaks Caller-sensitive Policies

Category	Severity	Location	Status
Logical Error	Medium	TakeRouter.sol:240-242	Resolved

Description

The renewal flow is documented and typed so ratifiers can evaluate the original orchestrator caller, not just the user being renewed. `IIntentRatifier.onIntentRatify` explicitly defines its `caller` argument as the address that called the orchestrator boundary, and `RenewalIntentRatifier._ratifyRate` forwards that value into `IInterestRatePolicy.getRate(...)` so policies can implement caller-sensitive authorization or pricing.

Vulnerability: the `ORCHESTRATOR_TAKE` branch in `TakeRouter` breaks that contract. Instead of preserving the external caller it calls `RenewalIntentTakeOnBehalf.take`, and `RenewalIntentTakeOnBehalf` then forwards its own `msg.sender` to the ratifier. For routed renewals, the ratifier therefore sees the router or adapter contract address instead of the real keeper / bundler initiator.

Impact: any deployment that relies on caller-sensitive `interestRatePolicy` logic can have that policy silently bypassed or unexpectedly DOSed when execution is routed through `TakeRouter`. `A policy that is supposed to whitelist specific keepers, deny specific operators, or return different rate curves per caller instead receives the router address for every routed renewal, collapsing distinct operators into one shared identity.`

The currently included `StaticRatePolicy` and `PausableStaticRatePolicy` ignore `caller`, which limits immediate impact for those implementations, but the bug defeats a security property the ratifier interface and comments explicitly advertise.

Recommendation

Clarify and enforce a single caller model across renewal execution. If caller-sensitive policies are not intended, document that `interestRatePolicy` must be caller-insensitive and do not treat caller as a security boundary. If caller-sensitive policies are intended, implement that behavior in a separate router-specific entrypoint rather than changing the existing `take` flow in place.

Resolution

Tenor Team: The issue was resolved in PR#407.

M-07 | onFlashLoan Missing Approval Breaks Bundled Ops

Category	Severity	Location	Status
Logical Error	Medium	MidnightAdapterBase.sol	Resolved

Description

`midnightFlashLoan` approves `assets` to Midnight before calling `MORPHO_MIDNIGHT.flashLoan`, then relies on that allowance surviving the callback so Midnight can pull the repayment on return:

```
    // Midnight.flashLoan
    SafeTransferLib.safeTransfer(token, msg.sender, assets);                // send to adapter
    IFlashLoanCallback(callback).onFlashLoan(token, assets, data);          // run nested bundle
    SafeTransferLib.safeTransferFrom(token, msg.sender, address(this), assets); // pull back via allowance
```

The adapter's callback forwards into the nested bundle and returns without restoring the allowance:

```
    function onFlashLoan(address, uint256, bytes memory data) external {
    require(msg.sender == address(MORPHO_MIDNIGHT), ErrorsLib.UnauthorizedSender());
    _midnightCallback(data);
    // no re-approval — Midnight will pull `assets` on return
    }
```

Any adapter action dispatched inside the nested bundle that `forceApprove`s Midnight for the same token will overwrite the outer allowance, and Midnight will consume whatever is left on its next pull. Same-token culprits: `midnightRepay`, `midnightSupplyCollateral` (when collateral token equals the flashLoan token), a nested `midnightFlashLoan`, and an inner `take` that routes onBuy through this adapter. In every case the outer `safeTransferFrom` sees allowance 0 and reverts, unwinding the bundle.

`onBuy` in the same file demonstrates the correct pattern — it re-approves Midnight for the exact pull amount after the nested bundle runs. `onFlashLoan` has identical post-callback pull semantics but does not mirror it.

Impact: the canonical flash-loan flows (flashLoan → repay debt → withdraw collateral → swap → repay, or flashLoan → take an offer → repay) are unreachable whenever the flashLoan token is also touched inside the nested bundle.

Recommendation

Restore the outer allowance at the end of `onFlashLoan`, mirroring `onBuy`. The callback already receives `token` and `assets`:

```
    function onFlashLoan(address token, uint256 assets, bytes memory data) external {
    require(msg.sender == address(MORPHO_MIDNIGHT), ErrorsLib.UnauthorizedSender());
    _midnightCallback(data);
    SafeERC20.forceApprove(IERC20(token), address(MORPHO_MIDNIGHT), assets);
    }
```

Resolution

Tenor Team: The issue was resolved in PR#406.

M-08 | Incorrect Math Inversion For SELL + Fee

Category	Severity	Location	Status
Rounding	Medium	src/periphery/libraries/ClampLib.sol:182	Resolved

Description

`ClampLib.maxUnitsForSellerBudget()` uses `mulDivDownInverse` (floor inverse) to compute the maximum units for SELL offers with `feeRate > 0`.

However, the actual repay budget uses ceiling arithmetic (`mulDivUp` via `sellerFeeFromTick`). When `units × effPrice % WAD ≠ 0`, the floor inverse returns a unit count one too high — causing `repayBudget = sourceDebt + 1`, which triggers an `ExcessRepayment` revert in Midnight.

This DoS affects fee-enabled full renewals via TakeRouter for SELL offers in the vast majority of real-world cases; the only safe inputs are those where `units × effPrice` is exactly divisible by WAD (1e18), which is coincidental in practice.

Recommendation

Replace `mulDivDownInverse` with a plain floor division in the SELL+feeRate>0 branch of `maxUnitsForSellerBudget`:

```
    // Before (wrong):
    return mulDivDownInverse(maxBudget, WAD, effPrice);
    // After (correct):
    return maxBudget.mulDivDown(WAD, effPrice);
```

Resolution

Tenor Team: The issue was resolved in PR#421.

M-09 | Executor Hides Real Liquidator From Gate

Category	Severity	Location	Status
Validation	Medium	MidnightVaultExecutor.sol	Resolved

Description

`MidnightVaultExecutor.liquidateAndRedeem()` calls `Midnight.liquidate()` as the executor contract, so the obligation's `liquidatorGate` only sees `address(MidnightVaultExecutor)`. The function is public. It does not check whether the external caller would pass the same gate.

This matters for caller-aware gates such as `MidnightAllowlistGate`. If the gate allowlists the executor so vault-share liquidations can use `liquidateAndRedeem()`, any external account can call the executor and inherit that permission. `Midnight.liquidate()` accepts the call because `canLiquidate(msg.sender)` is evaluated against the executor. The executor then forwards the seized value to the original caller by encoding `msg.sender` as the `liquidator` and redeeming seized shares to that address in `onLiquidate()`. The caller still has to fund `repaidUnits`, either by pre-funding the executor or by returning the repayment amount in its callback, but the caller-level liquidation restriction is gone.

This is an authorization bypass in a documented executor-based liquidation setup. The project documentation explicitly describes `liquidateAndRedeem()` as the liquidation mechanism where only the executor needs to be allowlisted and liquidators receive assets directly. If a market also relies on `liquidatorGate` to restrict which end liquidators may act, that policy is no longer enforced once the public executor is admitted. An unapproved liquidator can take the liquidation opportunity and spread even though the gate would reject that same account on a direct Midnight call.

Recommendation

Do not treat `MidnightVaultExecutor` as a safe public router for obligations that rely on caller identity at `liquidatorGate`.

Consider mirroring the gate check on the external caller before forwarding the liquidation. If `obligation.liquidatorGate != address(0)`, `liquidateAndRedeem()` should require `ILiquidatorGate(obligation.liquidatorGate).canLiquidate(msg.sender)` and revert otherwise.

Resolution

Tenor Team: The issue was resolved in PR#408.

M-10 | Zero Oracle Price Gives Free Collateral

Category	Severity	Location	Status
Unexpected Behavior	Medium	Midnight.sol	Acknowledged

Description

`Midnight.liquidate()` allows the liquidator to specify `seizedAssets` instead of `repaidUnits`. In that mode, the function computes the repayment from the selected collateral oracle price:

```
    repaidUnits = seizedAssets.mulDivUp(liquidatedCollatPrice,
ORACLE_PRICE_SCALE).mulDivUp(WAD, lif);
```

If the selected collateral oracle returns zero, `repaidUnits` becomes zero for any positive `seizedAssets`. The function still subtracts the collateral from the borrower, adds zero to `withdrawable`, transfers the seized collateral to the liquidator, and finally pulls zero loan tokens from the liquidator.

The same liquidation can also mark the borrower's debt as bad debt and update `lossIndex`, so lenders can be slashed while the liquidator receives collateral for no repayment. Tenor's `OracleWithValidationCheck` rejects a zero primary price, but Midnight obligations accept arbitrary oracle addresses. Any direct obligation or non-wrapper oracle that can transiently return zero exposes this path.

Recommendation

Reject zero selected-collateral prices whenever collateral can be seized. For example, in `liquidate()`:

```
    if (seizedAssets > 0 && liquidatedCollatPrice == 0) revert ZeroCollateralPrice();
```

More conservatively, require every active collateral oracle used in liquidation to return a nonzero price. Also document that a zero price is not just a liveness failure; in seized-asset liquidation mode it can become a value-extraction path.

Resolution

Tenor Team: Acknowledged.

M-11 | Keeper Params Do Not Guarantee Profitability

Category	Severity	Location	Status
Logical Error	Medium	src/BaseRenewalRatifier.sol	Acknowledged

Description

Tenor collects a callback fee on every renewal, sent to a protocol-controlled `feeRecipient`. There is no on-chain payment to the keeper. Tenor must run and subsidise the keeper itself.

The fee scales with position size and interest discount (`fee = feeRate × (1 - price) × units`), while keeper gas is fixed per renewal. For short renewal durations or small positions the fee falls below gas cost (around 415k gas spent on renewal). Adding more collaterals make it worse — each adds roughly 93k ($0.47 at 2 gwei) with no effect on the fee.

Measured gas (2 gwei, $2,500/ETH): $2.08 for 1 collateral, $3.94 for 5 collaterals.

At current market rates (4–5% APR), a $1,000 position renewed weekly earns $0.50 in fees per renewal against $2.08–$3.94 in gas — a $1.58–$3.44 loss per execution. Even a monthly renewal at 5% APR on a $1,000 position earns only $2.11, leaving no margin for gas spikes or additional collaterals. No minimum position size or renewal duration is enforced.

Recommendation

Document the break-even threshold explicitly (position size, renewal duration, collateral count, target chain) so keeper can make an informed decision. If self-sustaining keeper economics are required, enforce limits so `fee ≥ gas_cost` before executing.

Resolution

Tenor Team: Acknowledged.

M-12 | Temporary Grants Bypass Router Limits

Category	Severity	Location	Status
Validation	Medium	RenewalIntentTakeOnBehalf.sol	Acknowledged

Description

`RenewalIntentTakeOnBehalf.take()` has no action-scoped authorization and no reentrancy guard. It only checks that the user's ratifier is currently authorized, calls that ratifier, then forwards the take to Midnight.

This is unsafe when users rely on Bundler3 to grant renewal permissions temporarily. `AuthorizationAdapter` can grant `RenewalIntentTakeOnBehalf` and the user's renewal ratifier at the start of a multicall, run `TenorAdapter.execute()`, then revoke those permissions at the end. During the execution, those permissions are global.

`TakeRouter` only accounts for the amounts returned by the outer `_TAKE_ON_BEHALF.take()` call. If a malicious maker ratifier is called inside `Midnight.take()`, it can call `RenewalIntentTakeOnBehalf.take()` again for the same user while the temporary grants are still active. The nested renewal can pass the user's ratifier and Midnight checks, but it is invisible to the outer router. Therefore it is not included in `maxFill`, `minFill`, slippage checks, `BatchExecuted`, or `executeAndConsume` accounting.

For example, a lender submits a bundled renewal capped at `100` units and grants the needed permissions only for that bundle. The maker ratifier reenters during the first take and executes another valid renewal for `100` units before the outer router receives a result. The outer router then completes the original `100` unit fill and reports that amount. The user's position changed by `200` units even though the transaction-level router limit was `100`.

Recommendation

Do not use global boolean authorization as the only control for bundled renewals. Bind renewal authorization to a concrete execution scope that includes the expected caller, callback, source market, target market, expiry or nonce and maximum units. Consume that authorization before calling any untrusted ratifier or callback.

As an immediate hardening step, add a reentrancy guard to `RenewalIntentTakeOnBehalf.take()` so a maker ratifier, callback, token hook, or other synchronous external call cannot start another renewal before the first one returns. The router should also reject or explicitly account for nested orchestrator execution if nested renewals are intended to be supported.

Resolution

Tenor Team: Acknowledged.

M-13 | Flash Callback Spends Pending Reentry

Category	Severity	Location	Status
Validation	Medium	MidnightAdapterBase.sol	Acknowledged

Description

`MidnightAdapterBase.onFlashLoan()` forwards any Midnight flash-loan callback data to `Bundler3.reenter()` after only checking that `msg.sender` is `MORPHO_MIDNIGHT`. It does not record that `TenorAdapter` started the flash loan through `midnightFlashLoan()`. It also does not check the expected token, amount, callback data hash, offer, or router action before forwarding the reentry data.

`Midnight.flashLoan()` lets the caller choose both the callback address and the callback data. It also accepts `assets == 0`. Therefore, while Bundler3 is executing `TenorAdapter.execute()` with a pending `callbackHash = keccak256(reentryData)`, a malicious maker ratifier can call:

```
MORPHO_MIDNIGHT.flashLoan(token, 0, address(TenorAdapter), reentryData);
```

Midnight then calls `TenorAdapter.onFlashLoan(token, 0, reentryData)`. The adapter forwards `reentryData` to Bundler3. Bundler3 accepts the reentry because `TenorAdapter` is the caller and the reentry bytes match the pending hash. This consumes the user's precommitted reentry before the intended callback uses it.

The ratifier must know the exact reentry bytes. This can happen if the route passes those bytes in `ratifierData`, embeds them in maker-controlled offer data, or uses predictable callback data. The attacker is not granted arbitrary Bundler execution. Bundler3 still executes the user's precommitted calls, but it executes them at the wrong time.

This matters because `TenorAdapter` is used as a temporary payer inside a bundled transaction. For a direct `MIDNIGHT_TAKE` of a sell offer with no taker callback, `TakeRouter._dispatchMidnightTake()` gives Midnight a temporary allowance from `TenorAdapter`. Midnight also treats `TenorAdapter` as the payer. If the premature reentry withdraws the victim's source credit into the adapter before settlement, the resumed `Midnight.take()` can pull those tokens from the adapter and pay them to the maker.

The impact is stronger when the direct `MIDNIGHT_TAKE` targets an untrusted Midnight obligation. Obligations are created permissionlessly and `touchObligation()` only checks structural fields such as collateral ordering, allowed LLTV and `maxLif`. It does not whitelist collateral tokens or oracles. A malicious maker can publish a sell offer whose loan token is the real loan token, but whose collateral is a worthless attacker token and whose oracle is controlled by the attacker.

In that scenario, the premature reentry withdraws the victim's source credit to `TenorAdapter` before the attacker offer settles. Midnight then pulls the newly withdrawn loan tokens from the adapter and pays the maker. The victim receives credit in the attacker's obligation instead of keeping the original source credit.

The attacker-controlled oracle can report a high price during settlement so the seller health check passes. After settlement, the oracle can report zero. A zero-repay liquidation then realizes the attacker's debt as bad debt, clears the attacker debt and slashes the victim's toxic credit through the obligation loss index.

This requires a direct `MIDNIGHT_TAKE` sourced from an untrusted maker or obligation, exposed or predictable reentry bytes, withdrawable source credit and passing route price checks. It does not affect passive users. It also does not affect users routed only through canonical `ORCHESTRATOR_TAKE` execution with ratifier market-id binding. If no adapter reentry hash is pending, the malicious flash-loan callback cannot satisfy Bundler3's reentry check and the take reverts.

Recommendation

Only allow `onFlashLoan()` during a flash loan started by `midnightFlashLoan()`. Set a transient guard before calling `MORPHO_MIDNIGHT.flashLoan(token, assets, address(this), data)`. Store the expected token, amount and callback data hash. Require those values in `onFlashLoan()` before calling `_midnightCallback(data)`, then clear the guard after the flash loan returns.

Also bind adapter reentry data to the callback that is allowed to use it. The adapter should verify an expected callback selector, obligation id, taker role, subject address and offer or action hash before forwarding the inner `Call[]` to Bundler3.

For router-level defense in depth, raw `MIDNIGHT_TAKE` execution should reject offers whose obligation id or market id is not explicitly expected. The check should bind `offer.obligation` to an allowlisted target market or to the same market id authorized by the user's renewal intent.

Resolution

Tenor Team: Acknowledged.

M-14 | Source Unlocked During Borrow Renewal

Category	Severity	Location	Status
Reentrancy	Medium	RenewalIntentRatifier.sol, Midnight.sol,	Acknowledged

Description

`Midnight.take()` only applies the transient liquidation lock to the seller on the obligation being traded. In a V2-to-V2 borrow renewal, that traded obligation is the target obligation.

The source obligation is different, but `MorphoV2ToV2BorrowRepay.onSell()` later repays that source debt and moves source collateral. When the matched offer has `offer.buy == true`, the maker is the buyer and `offer.callback` runs as the buyer callback before the user's seller callback. The canonical renewal ratifier validates the user's renewal callback data, but it does not reject a nonzero maker-side buyer callback. Consequently, a maker can sign a BUY offer whose callback touches the user's source obligation before the renewal callback repays it. If the source position is already unhealthy, past maturity, or otherwise eligible for liquidation-related actions, the maker callback can call `DelayedLiquidationGate.startGracePeriod()` on the source obligation or partially liquidate the source before the renewal repairs it. A full source liquidation usually makes the later repayment revert because `repayBudget` can exceed the remaining source debt. The practical profitable case is a partial liquidation that leaves enough source debt for `MorphoV2ToV2BorrowRepay.onSell()` to continue, letting the attacker capture liquidation premium inside the same renewal transaction. The grace-period case can also leave a source timer armed even if the renewal later makes the source healthy.

Recommendation

Lock every Midnight obligation that a borrow renewal will mutate, not only the traded target obligation. For V2-to-V2 borrow renewals, the source obligation should be locked for the borrower before `Midnight.take()` executes any buyer callback. If Midnight cannot support multi-obligation locks, the renewal ratifier or wrapper should reject nonzero maker-side buyer callbacks for borrow renewals where the user is the seller. An allowlist is also acceptable if only audited inert callbacks are permitted. Do not rely on the token reentrancy assumption as the mitigation, because this vector uses the normal buyer callback order.

Resolution

Tenor Team: Acknowledged.

M-15 | executeAndConsume Burns Keeper consumeGroup

Category	Severity	Location	Status
Logical Error	Medium	TakeRouterAdapterBase.sol	Resolved

Description

`executeAndConsume` is supposed to couple a routed fill with a `consumed` update for the user whose position is being filled. The batch itself is executed on behalf of `params.taker`: `TakeRouter` authorizes the caller as `params.taker` or a Midnight-approved delegate, and the downstream takes use `params.taker` as the taker identity.

However, the final `setConsumed` call uses `initiator()` instead of `params.taker`:

```
    function executeAndConsume(ExecuteParams calldata params, Action[] calldata
actions, bytes32 consumeGroup)
    external
    onlyBundler3
    ...
    return (totals[0], totals[1], totals[2]);
    }
```

• A user authorizes a keeper on Midnight and relies on `executeAndConsume` to fill a self-limiting order group.

• The keeper calls the adapter with `params.taker = user`.

• The batch moves the user's position and returns non-zero `rawTotals`.

• The adapter updates `consumed` for the keeper's namespace, leaving `consumed[user][consumeGroup]` unchanged.

As a result, the user's order/group remains live even though the routed fill already happened. Any flow that relies on `consumeGroup` to cap or one-shot a user-owned order can therefore be overfilled or replayed through delegated execution.

Recommendation

Replace `initiator()` with `params.taker` in the final `setConsumed` path.

Resolution

Tenor Team: The issue was resolved in PR#418.

M-16 | Keeper Adapter Callbacks Reject Delegation

Category	Severity	Location	Status
Logical Error	Medium	MidnightAdapterBase.sol	Resolved

Description

`TakeRouterAdapterBase` explicitly supports delegated execution: the bundle submitter can differ from `params.taker` as long as the taker has authorized that caller on Midnight. This means a keeper can submit a Bundler3 bundle on behalf of a user and still pass the router's access control.

However, the adapter's own Midnight callbacks reject that same delegated shape. `onBuy` and `onSell` require the Midnight `buyer` or `seller` to equal `initiator()`:

```
    function onBuy(...) external returns (bytes32) {
    require(msg.sender == address(MORPHO_MIDNIGHT), ErrorsLib.UnauthorizedSender());
    require(buyer == initiator(), ErrorsLib.UnauthorizedSender()); // bundle submitter must be buyer
    ...
    }
    For `MIDNIGHT_TAKE`, Midnight passes the actual take ... into those callback roles, not the delegated caller:
```

// sell offer: taker is buyer

(

taker,

takerCallback,

takerCallbackData,

offer.maker,

offer.callback,

offer.callbackData,

offer.receiverIfMakerIsSeller

)

```
    As a result, any route that uses `TenorAdapter` itse ... r == params.taker`, and the adapter callback reverts.
```

Recommendation

`buyer/seller` should be validated against the taker authorization model, not strict equality with `initiator`. Mirror authorization in TakeRouter:

```
    address caller = initiator();
    require(
    buyer == caller || MORPHO_MIDNIGHT.isAuthorized(buyer, caller),
    ErrorsLib.UnauthorizedSender()
    );
```

Resolution

Tenor Team: The issue was resolved in PR#419.

M-17 | Adapter Callbacks Cannot Batch Reentries

Category	Severity	Location	Status
Compatibility	Medium	Bundler3.sol, TakeRouter.sol, MidnightAdapterBase.sol	Resolved

Description

`TakeRouter` is designed to fill a user's position across multiple actions in one call. `TenorAdapter` exposes that router through Bundler3, and its Midnight callbacks reenter Bundler3 through `_midnightCallback`.

The problem is that Bundler3 only supports one expected reentry per top-level call. Before each call in `multicall`, it stores a single `reenterHash` derived from that call's `callbackHash`:

```
    for (uint256 i; i < bundle.length; ++i) {
    address to = bundle[i].to;
    bytes32 callbackHash = bundle[i].callbackHash;
    ...
    require(reenterHash == bytes32(0), ErrorsLib.MissingExpectedReenter());
    }
```

That is incompatible with a single `adapter.execute` call that dispatches multiple `MIDNIGHT_TAKE` actions where more than one action uses `TenorAdapter` as its taker callback. `TakeRouter` loops through all actions inside the same top-level call and the first adapter callback consumes the only pending reentry:

```
    function onSell(...) external returns (bytes32) {
    require(msg.sender == address(MORPHO_MIDNIGHT), ErrorsLib.UnauthorizedSender());
    require(seller == initiator(), ErrorsLib.UnauthorizedSender());
    _midnightCallback(data); // consumes the only pending reenterHash
    return CALLBACK_SUCCESS;
    }
```

After that callback returns, `reenterHash` is cleared. If a later action in the same `adapter.execute` call reaches another adapter callback, `_midnightCallback` calls `Bundler3.reenter` again with no matching pending hash. Bundler3 reverts with `IncorrectReenterHash`, causing the action or the whole batch to fail.

A concrete example:

• Alice submits one `adapter.execute` batch with two actions, filling Bob's offer and Carol's offer.

• Both actions set `takerCallback = address(TenorAdapter)` because each fill needs a reentrant Bundler3 funding step.

• Bundler3 stores one `reenterHash` for the outer `adapter.execute` call.

• Bob's fill reaches `TenorAdapter.onSell`, calls `Bundler3.reenter`, and consumes that hash.

• Carol's fill later reaches `TenorAdapter.onSell` in the same router loop, calls `Bundler3.reenter` again, and reverts because `reenterHash` is already zero.

Recommendation

Consider a redesign of Tenor adapter operations around Bundler3's one-reentry-per-top-level-call model.

Resolution

Tenor Team: The issue was resolved in PR#420.

M-18 | executeAndConsume Accepts Intervening Fills

Category	Severity	Location	Status
Unexpected Behavior	Medium	TakeRouterAdapterBase.sol	Acknowledged

Description

`executeAndConsume()` executes the whole router batch before it reads the user's current `consumed` value for `consumeGroup`. It then writes `currentConsumed + rawTotals[params.fillIndex]`.

```
    (uint256[3] memory totals, uint256[3] memory rawTotals) = _executeResolvingSentinels(params, actions);
    address caller = initiator();
    _MIDNIGHT.setConsumed(
    consumeGroup, _MIDNIGHT.consumed(caller, consumeGroup) + rawTotals[params.fillIndex], caller
    );
```

This is unsafe when `executeAndConsume()` is used to implement an OCO-style action, where a user executes one route and wants the same transaction to consume or cancel a standing offer group. `_executeResolvingSentinels()` can call untrusted Midnight ratifiers, maker callbacks, taker callbacks, token transfers, and fee adjusters before the final `setConsumed()` call. Any of those external components can fill the user's live offer in `consumeGroup` before `executeAndConsume()` performs its final write.

Midnight's own `take()` checks the offer cap while it fills the offer. For a unit-capped offer, it increments `consumed[offer.maker][offer.group]` and requires the result to be at most `offer.maxUnits`. However, `setConsumed()` is a separate monotonic setter. It only requires the new value to be greater than or equal to the current value. It does not know the max value of any signed offer in that group, and it intentionally accepts values above the offer cap.

Consequently, a group can be filled once through the live offer and then advanced again by `executeAndConsume()` in the same transaction. For example, Alice has a standing sell offer in group `G` with `maxUnits = 100`. Alice submits `executeAndConsume(..., consumeGroup = G)` to sell `100` units elsewhere and mark `G` as consumed. During the router execution, a malicious maker callback fills Alice's standing offer for `100` units. The standing offer's own fill is valid, so Midnight sets `consumed[Alice][G] = 100`. The router then completes Alice's other `100` unit fill. Finally, `executeAndConsume()` reads the now-current consumed value and sets `consumed[Alice][G] = 200`.

The transaction does not revert even though Alice's standing offer was capped at `100` units. Alice experiences both fills. The impact is not arbitrary theft because the intervening fill must use an offer Alice already made available, and the router fill must also pass Alice's submitted route constraints. The broken property is the OCO guarantee: one fill is supposed to consume the shared capacity before the other can execute. A malicious counterparty, ratifier, or callback reached by the route can force both sides to execute under realistic conditions. The same race exists across transactions when the live offer is filled in the mempool before Alice's `executeAndConsume()` transaction lands, because the adapter has no `expectedConsumed` input.

Recommendation

Make `executeAndConsume()` bind the final write to an expected starting `consumed` value. The caller should provide `expectedConsumed`, and the adapter should check it before any external execution. After the router returns, the adapter should require that the same `consumeGroup` is still unchanged before writing the final value.

```
    function executeAndConsume(
    ExecuteParams calldata params,
    Action[] calldata actions,
    ...
    return (totals[0], totals[1], totals[2]);
    }
```

If the intended behavior is hard reservation rather than exact OCO accounting, pre-consume `expectedConsumed + maxFill` before external calls and then document that unused capacity is burned. If delegated execution remains supported, apply the same snapshot to the account whose group is being consumed.

Resolution

Tenor Team: Acknowledged.

M-19 | Clamps Ignore Revoked Callback Auth / Allowances

Category	Severity	Location	Status
DoS	Medium	VaultWithdrawClamp.sol, V2ToV2LendWithdrawableClamp.sol, V1ToV2LendClamp.sol,	Acknowledged

Description

Several withdrawal-backed clamps size actions from balances, debt, or withdrawable liquidity only, but they do not verify whether the callback still has the authorization or token allowance required to execute the action.

Examples:

• `VaultWithdrawClamp` assumes the callback can withdraw that collateral from Midnight on the user's behalf.

• `V2ToV2LendWithdrawableClamp` assumes the callback is still authorized to call `Midnight.withdraw(...)`.

• `V1ToV2LendClamp` does not check whether the vault-share allowance to `MorphoV1ToV2LendCallback` is still in place.

Those assumptions are revocable. A user can:

• revoke Midnight authorization for the callback contract

• revoke the callback's ERC20 / ERC4626 allowance

• leave an old offer signed and otherwise valid

After that, the clamps still report the offer as fillable because they only inspect economic state. But the real callback path reverts when it attempts the privileged operation.

The clamp output a likely offchain executability signal for production keepers, especially on callback-heavy routes where the callback-specific clamp is the intended way to avoid reverts. Because of that, the issue creates a real griefing vector:

1. A user signs a callback-backed offer and leaves it visible to the keeper network.
2. The same user later revokes the callback authorization or allowance needed by the callback path.
3. The keeper still sees a positive clamp result, because the clamp only checks economic state.
4. The keeper routes the action onchain.
5. The callback reverts on the missing auth / allowance.
6. If the action is batched with `allowRevert = false`, the entire batch aborts, including unrelated fills for other users.

The attacker cost is low: they do not need to break pricing or capital constraints, only to publish a seemingly valid signed offer and then revoke the callback permission that the clamp forgot to model.

The impact is an actionable DoS / gas-griefing vector against keeper infrastructure:

• takers and keepers can be lured into reverted fills

• routed batches can be deterministically reverted when `allowRevert` is not set

• unrelated users in the same keeper batch can be denied execution

• off-chain systems that treat clamp output as a reliable executability signal will overestimate available liquidity

Recommendation

For each clamp, include the callback's required authority in the view check when that authority is cheap to read on-chain.

Resolution

Tenor Team: Acknowledged.

M-20 | Tenor Prioritizes Midnight's Trading Fees

Category	Severity	Location	Status
Logical Error	Medium	src/callbacks/MorphoV2ToV2BorrowRepay.sol	Acknowledged

Description

Tenor callback fees are computed from the actual asset amount passed by Midnight into the callback, but that amount may already include Midnight's target obligation trading-fee adjustment. As a result, Midnight trading fees can consume the Tenor fee spread and cause a configured nonzero Tenor callback fee to settle as zero.

This was originally observed in `MorphoV2ToV2BorrowRepay.onSell()`. For BUY offers, Midnight deducts the target obligation trading fee from the borrower's `sellerAssets` before calling the borrower-side seller callback. The callback then compares that already-reduced `sellerAssets` against a seller effective price derived only from `offer.tick` and the Tenor fee rate. When the target Midnight trading fee is larger than the Tenor fee spread, `CallbackLib.sellerFeeFromTick()` zero-floors the fee to `0`. The source debt is still repaid and collateral is migrated, so the renewal succeeds while the configured Tenor fee recipient receives nothing.

The same issue also affects buyer-side lender flows. In `MorphoV2ToV2LendWithdrawable.onBuy()` and `MorphoV1ToV2LendCallback.onBuy()`, Midnight passes `buyerAssets` after adding the target obligation trading fee. Both callbacks compute the Tenor fee with `CallbackLib.buyerFeeFromTick(..., buyerAssets)`, which compares the Tenor buyer budget against the already-inflated `buyerAssets` and zero-floors the result. If the Midnight trading fee exceeds the Tenor buyer-side fee spread, the lender renewal or migration still succeeds and the lender receives target credit, but the Tenor fee recipient receives no fee.

Recommendation

Define the intended Tenor fee base explicitly and make callback fee calculation trading-fee-aware. If Tenor fees should apply independently of Midnight trading fees, compute Tenor fees from the tick-priced raw asset amount before Midnight target trading fees are applied, or pass both raw tick assets and trading-fee-adjusted settlement assets into the callback.

Do not compute Tenor fees solely from `sellerAssets` or `buyerAssets` after Midnight trading-fee adjustment. If Midnight trading fees intentionally take priority, document that nonzero Tenor fee configs can settle with zero callback fee across both borrower-side and lender-side renewal flows.

Resolution

Tenor Team: Acknowledged.

M-21 | Keeper Gets Full Midnight Control

Category	Severity	Location	Status
Access Control	Medium	src/periphery/TakeRouter.sol	Acknowledged

Description

Normal renewal execution through `RenewalIntentTakeOnBehalf.take()` does not require the user to authorize a keeper on Midnight. The renewal orchestrator is permissionless at the caller layer: any caller can submit a valid renewal, while the user-controlled protections come from `user -> takeOnBehalf` authorization on Midnight and `user -> ratifier` authorization on the renewal orchestrator. The ratifier then enforces the source/target markets, maturity window, rate policy, callback schema, and fee config.

The delegated Router path adds a different requirement. `TakeRouter._execute()` allows a non-user caller only when `Midnight.isAuthorized(params.taker, caller)` is true. For direct Router execution this means the user must authorize the keeper address. For `TenorAdapter`, `_caller()` resolves to the Bundler initiator, so the same condition applies to the keeper/initiator. The Router or adapter must also be authorized on Midnight so it can call `Midnight.take(... taker = user ...)`. That keeper authorization is not scoped to Router or renewal execution. It is full Midnight operator authority. Once granted, the keeper does not need Router at all: it can call Midnight directly as the user, bypassing `RenewalIntentTakeOnBehalf` and all renewal-ratifier checks. A malicious or compromised keeper can accept arbitrary valid offers as the user, choose `receiverIfTakerIsSeller`, withdraw the user's credit to any receiver, withdraw collateral while health checks allow, cancel offers via `setConsumed` or `shuffleSession`, and grant further authorization with `setIsAuthorized`.

This is a privilege-scope mismatch. Users may reasonably understand a keeper approval as limited automation for renewal fills, but the approval needed by the Router keeper path gives the keeper direct control over the user's Midnight account.

Recommendation

Do not require users to grant keepers direct Midnight authorization for Router-based renewal automation. Route delegated execution through a scoped contract authorization instead: the user should authorize a contract that enforces renewal-only action types, ratifier checks, receiver constraints, and callback schemas before calling Midnight. Alternatively, add a scoped authorization layer in Midnight or the Router that distinguishes "can call Router renewal flow" from "can operate all Midnight account functions." If the intended model is that keepers are fully trusted Midnight operators, document this explicitly in user-facing integration docs and avoid presenting keeper approval as limited renewal automation.

Resolution

Tenor Team: Acknowledged.

M-22 | TenorAdapter Bypasses Midnight's Repay Auth

Category	Severity	Location	Status
Validation	Medium	src/bundler/MidnightAdapterBase.sol	Resolved

Description

Midnight's account authorization check trusts `msg.sender`: a caller can act for `onBehalf` only if `onBehalf == msg.sender` or `isAuthorized[onBehalf][msg.sender]` is true. Once a user authorizes `TenorAdapter`, Midnight correctly trusts calls from that adapter.

The adapter must therefore bind every forwarded account action to the real Bundler3 `initiator()`. However, `midnightRepay()` fails to do this and accepts arbitrary `onBehalf`, so any unrelated Bundler3 initiator can call `TenorAdapter.midnightRepay(sourceObligation, 1, 0, victim)`. Midnight sees `msg.sender == TenorAdapter`, so the victim's adapter authorization is enough for the repay to pass even though the actual bundle initiator is not authorized by the victim.

Impacts:

• exact V2-to-V2 borrow renewals can be reverted. A 1 wei adapter repay lowers live source debt while the signed renewal still uses the old exact `repayBudget`; `MorphoV2ToV2BorrowRepay.onSell()` then reverts with `ExcessRepayment`.

• exact V2-to-V1 borrow exits can be reverted. The same dust repay makes an old exact BUY fill cross the borrower from debt into tiny credit, causing `MorphoV2ToV1BorrowCallback` to revert on `PositionCrossing`.

• reduce-only BUY offers can be invalidated. If a previously exact debt-reducing fill now exceeds live debt by 1 wei, Midnight rejects it because the maker would increase credit despite `reduceOnly`.

• exact repayment liquidations can be invalidated. A liquidator transaction quoted with `repaidUnits == borrowerDebt` becomes stale after a 1 wei repay and reverts, so liquidation bots must re-read and requote before retrying.

Recommendation

Pin `midnightRepay()` to the current Bundler3 initiator, matching the rest of the account-mutating adapter surface

Resolution

Tenor Team: The issue was resolved in PR#432.

M-23 | Arbitrary Policies Can Gas Grief Keepers

Category	Severity	Location	Status
DoS	Medium	src/RenewalIntentTakeOnBehalf.sol	Acknowledged

Description

A malicious user can install an arbitrary `interestRatePolicy`, and `BaseRenewalRatifier` will call it on every renewal through `getRate`.

`The policy could perform arbitrarily expensive read-only work before returning or reverting. Users fully control the policy address, and the docs describe StaticRatePolicy` and `PausableStaticRatePolicy` as examples rather than an allowlisted set.

A malicious policy can:

• burn substantial gas through expensive read-only logic before reverting,

• selectively reverts based on `caller`, `block.timestamp`, or other execution context,

• passes one offchain simulation and fails during real inclusion when the environment changes slightly.

This is more impactful than ordinary revoked auth / allowance grief. Those failures are typically cheap; here the router only catches the failure after the policy has already consumed most of the gas forwarded (EIP-150 63/64 rule) into `_TAKE_ON_BEHALF.take`. Hence, `allowRevert = true`is ineffective here as most of the gas budget is lost, preventing subsequent actions from continuing.

Recommendation

Consider capping the gas forwarded per `takeOnBehalf` action in `TakeRouter`(or alternatively at the `getRate` step) so that a single bad policy does not drain the keeper's remaining gas budget and `allowRevert` remains effective in practice.

More defensively, consider allowlisting approved `interestRatePolicy` implementations, or requiring policies to come from a trusted factory.

Resolution

Tenor Team: Acknowledged.

M-24 | Permissionless First Touch Freezes Default Fees

Category	Severity	Location	Status
Unexpected Behavior	Medium	Midnight.sol	Acknowledged

Description

`touchObligation()` can be called by anyone and the first caller copies the current default trading and continuous fees into the obligation. Later calls to `setDefaultTradingFee()` or `setDefaultContinuousFee()` do not affect obligations that were already touched. The fee setter can repair a known touched obligation with `setObligationTradingFee()` and `setObligationContinuousFee()`, but default fee updates will not repair it automatically.

Consequently, a searcher can pre-create predictable future obligations before the fee setter raises defaults. Those obligations keep the old fee schedule even after governance expects new markets for the same loan token to use higher fees. Users and keepers can then trade against the pre-created obligations and avoid both the trading fee and the continuous fee until each obligation is discovered and patched.

For a concrete Tenor case, assume USDC renewals are expected to use a known collateral set, oracle set, LLTV, gate configuration, and four-week maturity cadence. During launch, the default Midnight fees for USDC are still zero or temporarily lower. A searcher can enumerate the next several Tenor target obligations from those public parameters and call `touchObligation()` for each future maturity. When Morpho later raises the USDC default trading and continuous fees, those pre-created target obligations keep the old schedule. A borrower or lender can then route renewals through `RenewalIntentTakeOnBehalf` into the pre-created obligations and settle at the intended Tenor market and maturity, but Midnight charges the stale fees stored on first touch. The same renewal into an otherwise identical target obligation created after the fee update would accrue protocol fees.

The obligations can be created by anyone, the future Tenor obligation ids are predictable from public market parameters and skipped fees are not recoverable. Any trading fee skipped before the obligation-specific repair is permanently lost. Continuous-fee liability for credit opened while the stale fee was active is also fixed from the old `continuousFee`. A later repair only applies to new credit opened after the patch. It does not retroactively charge the correct liability to already-opened credit, so existing credit keeps the stale fee value through maturity.

Recommendation

Do not let arbitrary callers lock in protocol defaults for future obligations. Either require an authorized fee setter or market creator to initialize fee-bearing obligations or make untouched obligations read current defaults until the first real position-changing action. If immutable per-obligation fees are required, Tenor should treat fee initialization as an explicit market-activation step. Before allowing renewals into a known obligation, compute its id, call `touchObligation()` if needed and have Midnight's `feeSetter` call `setObligationTradingFee()` for all seven trading-fee breakpoints plus `setObligationContinuousFee()` for that obligation. The ratifier can also reject target obligations whose stored `tradingFees(id)` or `continuousFee(id)` do not match the expected schedule.

This recovery only protects future trades after the obligation-specific fee update. It cannot recover trading fees already skipped. It also cannot retroactively assign the correct continuous-fee liability to credit that was opened while the stale fee was active.

Resolution

Tenor Team: Acknowledged.

M-25 | Fresh Credit Drains Old Repayments

Category	Severity	Location	Status
Validation	Medium	Midnight.sol, MorphoV2ToV2LendWithdrawable.sol	Acknowledged

Description

`Midnight` tracks repaid cash in one obligation-level `withdrawable` bucket. `repay()` adds units to that bucket, and `withdraw()` lets any current credit holder burn credit and pull from the bucket. There is no accounting that ties a repayment to the lenders whose credit existed when that repayment happened. That lets a new buyer take a discounted SELL offer, receive fresh credit, and immediately redeem that fresh credit against old repayments at par. For example, if an obligation already has `100e18` withdrawable units and a borrower sells `100e18` new units at a discounted price, the buyer can pay less than `100e18`, receive `100e18` credit, then call `withdraw()` for `100e18`. The buyer captures the discount while the old lenders' repaid liquidity is drained and replaced with exposure to the new borrower's debt.

This also explains the callback self-funding variants. `Midnight.take()` updates buyer credit before callbacks and before final payment settlement. If a callback path can call `withdraw()` on the same obligation, the fresh credit created by the in-flight take is treated as source credit for old `withdrawable` liquidity:

- `MorphoV2ToV2LendWithdrawable.onBuy()` can be called with `sourceObligation == obligation`, so direct callback data can withdraw `buyerAssets + fee` from the same obligation immediately after the buyer credit is minted.
- `MidnightAdapterBase.onBuy()` can reenter Bundler3 before approving Midnight to collect `buyerAssets`, and the reentered bundle can call `midnightWithdraw()` against the same obligation.

Those callback routes remove the buyer's upfront capital requirement, but they are not a separate economic root cause. They are same-transaction manifestations of the broader issue: newly created credit can claim older repayments from the global `withdrawable` pool. Even without a callback, the buyer can fund the discounted purchase with their own balance or external flash liquidity and withdraw old repayments after the take settles. If the offer uses a zero-price tick, the same entitlement bug becomes a zero-principal version of the attack. The existing zero-price cap issue covers the cap-accounting footgun, but the loss here is the ability of newly created credit to claim older repayments.

The root cause is that `take()` creates transferable/redeemable buyer credit immediately:

```
    uint256 buyerCreditIncrease = UtilsLib.zeroFloorSub(units, buyerPos.debt);
    uint256 sellerCreditDecrease = UtilsLib.min(units, sellerPos.credit);
    uint256 sellerDebtIncrease = units - sellerCreditDecrease;
    ...
    _obligationState.totalUnits =
    UtilsLib.toUint128(_obligationState.totalUnits + buyerCreditIncrease - sellerCreditDecrease);
```

Then `withdraw()` only checks the caller's current credit and the obligation's global `withdrawable` balance. It never checks whether that credit existed when the repaid liquidity entered the bucket:

```
    _updatePosition(obligation, id, onBehalf);
    Position storage _position = position[id][onBehalf];
    uint128 pendingFeeDecrease;
    ...
    _obligationState.totalUnits -= UtilsLib.toUint128(units);
    SafeTransferLib.safeTransfer(obligation.loanToken, receiver, units);
```

Recommendation

Make repayment entitlement position-indexed instead of first-come-first-served. A typical fix is to maintain a global withdrawal index that increases on repayment, track each position's index and initialize newly created credit at the current index so it cannot claim older repayments.

As a blunt mitigation, reject debt-increasing takes while an obligation has nonzero `withdrawable`, but that is likely too restrictive. The durable fix is to separate old repayment claims from newly minted credit.

Callback-specific guards are still useful defense in depth. `MorphoV2ToV2LendWithdrawable.onBuy()` should reject `sourceObligationId == targetObligationId`, and generic adapter buy callbacks should not be allowed to withdraw newly created same-obligation credit during the in-flight take. Those local guards reduce the no-capital self-funding paths, but they do not fix the post-settlement version where a buyer funds the discounted purchase and then withdraws old repayments at par.

Related follow-up issue: I-16.

Resolution

Tenor Team: Acknowledged.

L-01 | allowRevert Misses Pre-dispatch Reverts

Category	Severity	Location	Status
Unexpected Behavior	Low	TakeRouter.sol	Acknowledged

Description

`allowRevert` only catches failures from `_TAKE_ON_BEHALF.take()` and `_MIDNIGHT.take()`. Both dispatch helpers do meaningful work before reaching those `try/catch` blocks. They decode action data, call `_MIDNIGHT.touchObligation()` and call `_capTakeUnits()`. `_capTakeUnits()` then reaches `RouterLib.budgetToUnits()`, `ClampLib.getOfferRemaining()` and any caller-supplied fee adjuster or clamp.

Those operations can revert before the router has entered the soft-fail boundary. For example, `ClampLib.getOfferRemaining()` computes `offerPrice - tradingFee` for buy offers without a zero floor. A low-tick buy offer can therefore underflow before `_TAKE_ON_BEHALF.take()` or `_MIDNIGHT.take()` is attempted. Similarly, asset-denominated sizing can revert inside `TakeAmountsLib` before the action reaches the catch block.

Consequently a single malformed or adversarial offer can abort the entire batch even when its `Action.allowRevert` flag is set. This contradicts the documented soft-failure behavior and creates a griefing target for keepers that aggregate untrusted actions.

Recommendation

Move all maker-controlled pre-dispatch work inside the same failure-isolation boundary as the downstream take. A practical fix is to wrap obligation touching and unit capping in an internal call that can be caught, then convert any failure into `(false, 0, 0, 0, reason)` when `allowRevert` is enabled.

Also normalize known sizing edge cases, such as zero prices and `tradingFee > offerPrice`, so they return a safe zero cap where possible instead of panicking before the action can be skipped.

Resolution

Tenor Team: Acknowledged.

L-02 | Frozen Gate Trusts Fee Recipients

Category	Severity	Location	Status
Trust Assumptions	Low	VaultV2AllowlistGate.sol	Acknowledged

Description

`canReceiveShares` does not rely only on the gate's own `allowlist`. It also returns `true` for whatever addresses `msg.sender` currently exposes as `managementFeeRecipient()` and `performanceFeeRecipient()`. That means `renounceOwnership()` does not freeze the effective receive-share whitelist. `VaultV2` keeps `setManagementFeeRecipient` and `setPerformanceFeeRecipient` as live curator actions, so the curator can nominate any non-zero address after the gate owner has burned control. The new recipient becomes immediately eligible to receive shares even if it was never present in `allowlist`. This also works when both fee rates are zero, because the gate checks only the recipient fields.

This matters because the gate is documented as the mechanism that keeps vault shares away from secondary markets and lending venues. A curator can later reopen that surface for an arbitrary EOA or protocol without touching the gate mapping and without re-enabling `setReceiveSharesGate`. Consequently, the "immutable allowlist" guarantee is weaker than it appears, and the share-receive restriction can be relaxed after the setup is supposedly frozen.

Recommendation

Do not derive share-receive permission from live vault fee-recipient fields. Require fee recipients to be explicitly allowlisted, then freeze only the gate's own storage. If the auto-whitelist must remain, the deployment process should also abdicate both fee-recipient setters before the gate is treated as frozen and that requirement should be documented clearly.

Resolution

Tenor Team: The issue was resolved in PR#452.

L-03 | Vault-share Exit Ignores Vault Withdrawal Limits

Category	Severity	Location	Status
Validation	Low	MorphoV2WithdrawVaultSharesCallback.sol	Acknowledged

Description

`onBuy` sizes the collateral pull with `previewWithdraw(buyerAssets)` and then immediately calls `withdraw(buyerAssets, ...)`. That only proves how many shares are needed at the current exchange rate. It does not prove that `buyerAssets` is actually withdrawable from the vault.

ERC4626 explicitly separates `previewWithdraw` from withdrawal limits. `previewWithdraw` must ignore limits such as queues, cooldowns and liquidity ceilings, while `withdraw` must revert if the requested assets cannot actually be withdrawn. This callback never checks any executable withdrawal bound after it receives the buyer's shares, so a position can hold enough vault-share value on Midnight while the vault still refuses to redeem the requested assets. In that state the callback reverts inside `withdraw`, which makes the BUY offer unfillable.

This is a liveness issue on its own. It becomes a taker-griefing issue because `VaultWithdrawClamp` bounds fills with `convertToAssets(userVaultShares)` instead of redeemable liquidity. Routers and keepers can therefore derive a non-zero fill size for a callback execution that is guaranteed to revert once the vault enforces its withdrawal ceiling.

The intended VaultV2 integration has the same problem, but the right bound is not `maxWithdraw`: `VaultV2.maxWithdraw()` deliberately returns `0` because the vault cannot make a revert-free promise when gates may be installed. `VaultV2.withdraw() can still revert after the clamp returns a positive value because exit()` checks `canSendShares(onBehalf)` and `canReceiveAssets(receiver)`, then may call `deallocateInternal()` through the liquidity adapter. The Morpho Blue adapter withdraws exact assets from Morpho Blue and the MetaMorpho adapter calls the child vault's `withdraw()`. Both can fail when liquidity, queues, adapter state, or the underlying vault make the requested assets unavailable. The current clamp does not inspect any of those conditions.

Recommendation

Do not treat share balance as redeemable liquidity. For generic ERC4626 vaults, the callback should check the actual withdrawal bound, for example `maxWithdraw(address(this))`, after the shares are transferred in and before calling `withdraw`. Clamps should only advertise capacity when they have a reliable pre-transfer bound for the supported vault.

For VaultV2, do not rely on `maxWithdraw()`. The clamp should check that the vault exit can actually go through: the right gate permissions must be in place, and the liquidity adapter must be able to provide the requested assets. If the clamp cannot reliably tell how much is withdrawable, it should return zero for that vault type or clearly document that these fills may revert. It should not use `convertToAssets()` as if share value always means withdrawable liquidity.

Resolution

Tenor Team: Acknowledged.

L-04 | Zero-price Asset Caps Act Unbounded

Category	Severity	Location	Status
Unexpected Behavior	Low	Midnight.sol	Acknowledged

Description

`Midnight.take()` lets makers cap an offer by `maxUnits`, `maxSellerAssets`, or `maxBuyerAssets`. Unit-capped offers advance `consumed[maker][group]` by the filled units. Asset-capped offers instead advance `consumed[maker][group]` by the computed asset amount. That distinction breaks down when the selected asset amount is zero for a nonzero unit fill. `TickLib.tickToPrice(0)` returns `0`. For a SELL offer at that tick, `sellerPrice` is zero, so `sellerAssets` is also zero for any positive `units`.

```
    uint256 offerPrice = TickLib.tickToPrice(offer.tick);
    uint256 sellerPrice = offer.buy ? offerPrice - _tradingFee : offerPrice;
    uint256 buyerPrice = sellerPrice + _tradingFee;
    ...
    require(newConsumed <= offer.maxSellerAssets, ConsumedSellerAssets());
    }
```

For a SELL offer capped by `maxSellerAssets`, every nonzero fill transfers obligation units while adding zero to the cap counter. The signed asset cap never becomes exhausted in unit terms. If the seller already owns credit and marks the offer `reduceOnly`, takers can buy that existing credit for zero seller proceeds until the seller's credit is gone. The practical bound is the seller's remaining credit, not the finite asset cap.

This should be treated as a maker or integration footgun rather than arbitrary theft. The maker must authorize a zero-price asset-denominated offer, or an integration must create one on the maker's behalf. A maker who manually signs this offer has effectively agreed to a zero price. The unsafe part is that a finite asset cap may look like an exposure cap even though it cannot bound units when the selected asset leg is always zero.

Zero-price trading is also not impossible input in this codebase. Midnight's tests cover zero-price unit fills, and periphery helpers explicitly treat zero price as unbounded unit capacity. For example, `ClampLib.getOfferRemaining()` returns `type(uint128).max` when a finite seller-asset cap has a zero seller price. Consequently, router sizing can treat the same finite asset-capped offer as having effectively unbounded remaining unit capacity.

The cleanest case is a SELL offer with `tick == 0`, `maxSellerAssets > 0`, and no unit cap. This case does not depend on buyer-side trading fee for the cap itself because SELL `sellerAssets` uses `offerPrice` directly. If trading fee is nonzero, the taker may still pay the protocol fee, but the seller receives zero assets and `consumed` remains unchanged. In the no-fee case, both buyer and seller asset amounts are zero.

For example, Alice owns `100e18` units of credit on a Midnight obligation. She signs a reduce-only SELL offer with `tick = 0`, `maxSellerAssets = 1`, `maxUnits = 0`, and a fresh group. Bob fills `60e18` units. `Midnight.take()` emits `buyerAssets = 0`, `sellerAssets = 0`, and `units = 60e18`. Alice's credit falls by `60e18`, Bob's credit rises by `60e18`, and `consumed[Alice][group]` is still zero. Bob can then fill the same offer for another `40e18` units because the group cap still appears unused.

Recommendation

Document that zero-price asset-denominated caps do not bound unit exposure. Maker-facing integrations should reject this configuration unless the user explicitly opts into a free unit transfer. Reject asset-denominated offers when the selected asset leg can be zero for a positive unit fill. For the main SELL case, require `offerPrice > 0` whenever `offer.buy == false` and `maxSellerAssets > 0`. For BUY offers, require the relevant buyer or seller price to be positive before allowing the corresponding asset cap.

The periphery should also avoid reporting unbounded unit capacity for finite zero-priced asset caps unless that behavior is deliberately part of the public API. `ClampLib.getOfferRemaining()` can return `0` for such offers, or expose a distinct result that callers must handle explicitly.

Resolution

Tenor Team: Acknowledged.

L-05 | isFinalFill Never True With Non-Zero Fee

Category	Severity	Location	Status
Rounding	Low	src/callbacks/MorphoV2ToV2BorrowRepay.sol:53	Acknowledged

Description

`CollateralTransferLib` checks `isFinalFill = (sourceDebtBefore == repaidUnits)` to decide whether to sweep all source collateral. But `repaidUnits` is set to `sellerAssets - fee` in the callback, which is always strictly less than `sourceDebt` when `fee > 0` — `sellerFeeFromTick` uses ceiling arithmetic, so `fee ≥ 1` whenever interest rate > 0. The equality can never hold, and `isFinalFill` is permanently false for all fee-enabled renewals. As a result, every fee-enabled renewal leaves a residual debt (≈ fee in loan token units) and a proportional share of source collateral stranded in the source obligation. At standard parameters on a 1000 ETH position this amounts to ~4.67 ETH of residual debt and ~24.8 ETH of stranded collateral. The zero-fee path is unaffected.

Recommendation

Replace the static budget comparison with a post-repay on-chain state check:

```
    bool isFinalFill = MORPHO_MIDNIGHT.debtOf(sourceObligationId, seller) == 0;
```

Resolution

Tenor Team: Acknowledged.

L-06 | Fee Recipients Can Get Stuck Shares

Category	Severity	Location	Status
Unexpected Behavior	Low	VaultV2AllowlistGate.sol	Resolved

Description

The contract special-cases fee recipients only in `canReceiveShares`. `canSendShares` and `canReceiveAssets` still read the static `allowlist` with no equivalent exemption. When the same gate is installed as `sendSharesGate`, `VaultV2.accrueInterest()` can mint fee shares to a management or performance fee recipient that was never manually allowlisted. Those shares arrive successfully because `canReceiveShares` returns `true`, but any later `redeem` or `withdraw` from that recipient reverts because `VaultV2.exit()` requires `canSendShares(onBehalf)`. If `receiveAssetsGate` is also set, the recipient may fail that check as well. If ownership has already been renounced, there is no recovery path inside the gate. Fees can keep accruing and diluting other holders, while the corresponding claim stays stranded behind the exit gates. This turns a convenience exemption for fee minting into a permanent liveness failure for fee realization whenever `sendSharesGate` is active and fee recipients were not explicitly granted exit permissions.

Recommendation

Fee recipients need an exit permission set as well as a mint permission set. Either require operators to add explicit `canSendShares` and, when relevant, `canReceiveAssets` permissions for fee recipients before freezing the gate, or make the fee-recipient exemption configurable for the directions needed to redeem fees. If the intended deployment only supports `receiveSharesGate`, document that using this contract as `sendSharesGate` requires manual fee-recipient allowlisting.

Resolution

Tenor Team: The issue was resolved in PR#413.

L-07 | Clamps Ignore Active Collateral Cap

Category	Severity	Location	Status
Validation	Low	SupplyCollateralCallbackClamp.sol, V2ToV2BorrowRepayClamp.sol	Resolved

Description

`SupplyCollateralCallbackClamp.clamp()` sizes a `MorphoV2SupplyCollateralCallback` fill from the caller-supplied collateral amounts, the seller's token balances and allowances, current debt, and the projected health or raw max-LTV bound. It never checks whether the callback schedule would activate more collateral slots than Midnight allows for one borrower.

Midnight permits obligations with up to 128 collateral parameters, but `supplyCollateral()` reverts with `TooManyActivatedCollaterals()` once a borrower's activated bitmap would exceed `MAX_COLLATERALS_PER_BORROWER` (10). The supply callback calls `supplyCollateral()` once per nonzero scheduled slot. If the seller already has 10 active collateral slots, or if the callback schedule activates more than 10 fresh slots during a fill, the clamp can still return a positive max fill because it treats every nonzero `collateralAmounts[i]` as valid projected collateral. Execution then reverts part way through `onSell()` when the later slot is supplied.

The same mismatch exists for V2-to-V2 borrow renewals. `MorphoV2ToV2BorrowRepay.onSell()` calls `CollateralTransferLib.transferCollaterals()`, which withdraws matching source collateral and re-supplies it into the target obligation. `V2ToV2BorrowRepayClamp.clamp()` only caps by source debt and reduce-only target accounting. It does not model the union of the borrower's already active target collateral slots and the source collateral slots that would become newly active during migration. A borrower can therefore have a clamp-reported fill where the target already has 10 active slots and the migrated source collateral would activate an 11th target slot, causing `supplyCollateral()` to revert.

This does not let a taker bypass Midnight health or steal funds. The revert happens inside the same `Midnight.take()` call after the seller callback, so the prior debt updates, token transfers, approvals, and earlier collateral supplies roll back. The impact is liveness and keeper griefing: router users can receive a positive clamp quote for an offer that cannot execute under Midnight's real collateral-activation rule, and repeated `allowRevert` batches can burn gas on these doomed actions.

Recommendation

Make the clamps enforce the same activation-cap invariant before returning a nonzero fill. For `SupplyCollateralCallbackClamp`, read `MIDNIGHT.activatedCollaterals(data.obligationId, offer.maker)`, count the already active bits, then add each callback slot where the current collateral is zero and the pro-rata supply for the candidate fill would be nonzero.

For `V2ToV2BorrowRepayClamp`, either require source and target active-collateral compatibility before advertising a fill, or extend the clamp data so the clamp can compare the target active bitmap with the source collateral slots that `CollateralTransferLib` would migrate for the proposed fill. Return zero, or cap below the first nonzero activation, when the resulting target active slot count would exceed `MAX_COLLATERALS_PER_BORROWER`.

At minimum, offer builders should reject supply-collateral schedules and V2-to-V2 migration tuples that can activate more than 10 target slots for the borrower. The tighter fix is to put this check in the clamp layer so router sizing matches `Midnight.supplyCollateral()`.

Resolution

Tenor Team: The issue was resolved in PR#410.

L-08 | Fragmented Fills Skip Discrete Collateral

Category	Severity	Location	Status
Rounding	Low	MorphoV2SupplyCollateralCallback.sol	Acknowledged

Description

`onSell` computes each collateral deposit independently for each fill with `mulDivDown()`. If a nonzero configured collateral amount is small relative to `offerSellerAssets`, a taker can split the offer into fills where every individual `supplyAmount` rounds down to zero. The callback then skips the transfer and `supplyCollateral()` call for that slot, even though the same total fill executed in one transaction would have supplied collateral.

This affects seller-asset-denominated offers, not only unsupported `maxUnits` offers. For example, a callback schedule can require one raw unit of a 0-decimal collateral token for a full `100e18` seller-asset fill. If the taker instead fills the offer as one hundred `1e18` fills, each fill computes `1 * 1e18 / 100e18 = 0` raw collateral units. The taker creates the same aggregate debt while none of that configured callback collateral is supplied. This can remain healthy when the seller already has enough collateral, but the final collateralization differs from the signed callback schedule.

The final impact is configuration-dependent. It is most relevant for low-decimal collateral, small configured collateral amounts, loose or disabled `maxLtv` and accounts with enough existing collateral for dust fills to pass Midnight's final health check. It does not directly steal funds, but it lets a taker consume an offer while leaving the maker with less callback-supplied collateral than the maker would get from an equivalent full fill.

Recommendation

Do not rely on per-fill floor rounding for exact collateral schedules. Enforce a minimum fill size for every nonzero collateral slot so `amounts[i] * sellerAssets / offerSellerAssets` is nonzero whenever that slot is expected to contribute collateral. If exact aggregate collateral is required across arbitrary partial fills, add stateful residual accounting keyed by the offer or group so rounding dust carries into later fills. As a simpler operational mitigation, require meaningful `maxLtv` caps and reject low-decimal callback configurations where the configured amount is too small relative to the allowed minimum fill.

Resolution

Tenor Team: Acknowledged.

L-09 | Target-side Netting Changes Renewal Semantics

Category	Severity	Location	Status
Unexpected Behavior	Low	Midnight.sol, MorphoV2ToV2BorrowRepay.sol, BaseRenewalRatifier.sol, MorphoV2ToV2LendWithdrawable.sol	Acknowledged

Description

Midnight nets a user's existing opposite-side position on the target obligation before creating new credit or debt. Tenor's V2-to-V2 renewal callbacks assume the target fill opens or increases the expected target position, but `Midnight.take()` first consumes any existing opposite-side position.

For borrow renewals, the user is the seller on the target obligation. If the borrower already has credit on the target obligation, the fill consumes that credit before creating new target debt:

```
uint256 sellerCreditDecrease = UtilsLib.min(units, sellerPos.credit);
uint256 sellerDebtIncrease = units - sellerCreditDecrease;
sellerPos.credit -= UtilsLib.toUint128(sellerCreditDecrease);
sellerPos.debt += UtilsLib.toUint128(sellerDebtIncrease);
```

`MorphoV2ToV2BorrowRepay.onSell()` then repays source debt and migrates collateral as if the fill were a normal borrow renewal. Economically, the user's existing target lend was sold/netted to fund the migration.

For lend renewals, the user is the buyer on the target obligation. If the lender already has target debt, the fill reduces that debt before creating new target credit. `MorphoV2ToV2LendWithdrawable.onBuy()` still withdraws source credit to fund the BUY offer, so the route performs debt buyback rather than opening new target lending credit.

The ratifier checks market IDs, fee config, rate, maturity, cadence, and callback data, but it does not reject nonzero target opposite-side positions. `reduceOnly` does not protect the user here because it constrains the maker, not the taker-side renewal subject.

Consequently, a validated renewal can succeed while producing a different user-visible operation from the one configured:

• borrow renewal can spend existing target credit instead of creating target debt;

• lend renewal can spend source credit to pay down target debt instead of creating target credit;

• callback events and route labels can describe a renewal while the economic result is netting/buyback.

This is not necessarily unsafe when the user explicitly wants netting, but it should not be implicit in ordinary renewal routes.

Recommendation

For standard renewals, reject execution when the user has an opposite-side position on the target obligation:

• V2-to-V2 borrow renewal: reject if `updatePositionView(targetObligation, targetId, borrower).credit != 0`.

• V2-to-V2 lend renewal: reject if `debtOf(targetId, lender) != 0`.

If netting is intended, expose it as a separate route with explicit naming, separate user opt-in, and events that report how much target credit/debt was netted versus newly opened.

Resolution

Tenor Team: Acknowledged.

L-10 | Oracle Validation Can Fail Open Or Freeze

Category	Severity	Location	Status
Oracle	Low	OracleWithValidationCheck.sol	Acknowledged

Description

`OracleWithValidationCheck` can lose its secondary-validation property in several operational modes. When `REVERT_ON_VALIDATION_ORACLE_PRICE_FAILURE == false`, any revert from the validation oracle is caught and the primary price is returned without cross-checking. This includes deliberate validation-oracle circuit breakers. The wrapper can therefore become primary-only exactly when the validation oracle is signaling failure. Graceful mode is also incomplete: Solidity `try ... returns (uint256)` catches call reverts, but malformed successful return data can revert during ABI decoding and bypass the intended graceful fallback. The owner can pause validation, and only the owner can unpause. If ownership is renounced while paused, the wrapper is permanently primary-only. Conversely, in strict mode, renouncing ownership removes the only built-in recovery control for future validation-oracle failure. Consequently, depending on deployment mode, the oracle can silently run without validation, remain permanently primary-only, or become permanently unusable when the validation oracle fails.

Recommendation

Use strict mode for safety-critical markets unless primary-only fallback is explicitly accepted. If graceful mode is kept, use low-level `staticcall` and validate returndata length before decoding. Override `renounceOwnership()` so it cannot be called while validation is paused and document that strict immutable deployments intentionally give up emergency pause recovery.

Resolution

Tenor Team: Acknowledged.

L-11 | Vault Executor Redeems Without Asset Floors

Category	Severity	Location	Status
Validation	Low	MidnightVaultExecutor.sol	Resolved

Description

`MidnightVaultExecutor` exposes redeem operations without caller-supplied minimum asset amounts.

`withdrawCollateralAndRedeem()` withdraws an exact number of vault shares from Midnight and accepts whatever `redeem()` returns for the receiver. `onRepay()` does the same while trying to fund a Midnight repayment. `onLiquidate()` redeems the seized shares and forwards whatever assets the vault returns to the liquidator.

For a trusted stable vault this may be acceptable. As a public executor, though, the caller cannot protect against vault losses, share-price movement between transaction construction and execution, or supported vaults whose redeemable value changes because of queues, fees, donations, or adapter state. A withdrawal or liquidation can execute with materially fewer assets than the user or liquidator expected and the transaction has no local bound that can stop it.

Recommendation

Add explicit minimum asset parameters to redeeming entrypoints. `withdrawCollateralAndRedeem()` should require `assets >= minAssetsOut`. `repayAndWithdrawCollateral()` should require the redeemed amount to cover the requested repayment plus any user-specified surplus floor. `liquidateAndRedeem()` should let the liquidator provide a minimum redeemed amount or minimum liquidation surplus and revert when the actual redeem output is below that value.

Resolution

Tenor Team: The issue was resolved in PR#424.

L-12 | Fee Adjuster Overshoots maxFill

Category	Severity	Location	Status
Rounding	Low	src/periphery/resolvers/CallbackFeeAdjuster.sol	Resolved

Description

`CallbackFeeAdjuster.beforeDispatch()` is intended to cap `takeUnits` so the router's effective fill in `fillIndex` stays within the remaining budget. For SELL offers using percentage-fee adjustment with `fillIndex == FILL_BUYER_ASSETS`, the closed-form inverse can return a units cap that is one wei too large.

The router dispatches `Midnight.take()` with that cap, then `CallbackFeeAdjuster.afterDispatch()` adds the percentage fee to `buyerAssets`. On affected rounding boundaries, the adjusted total exceeds `maxFill` by one wei and `TakeRouter` reverts with `FillOvershoot`.

This does not create a value-stealing primitive because the revert unwinds the transaction. The impact is deterministic DoS of otherwise valid exact-budget routed fills, forcing keepers or offchain routers to add slack or avoid exact `FILL_BUYER_ASSETS` percentage-fee routes.

Recommendation

Add a forward-check correction after the inverse calculation for percentage-fee `FILL_BUYER_ASSETS` sizing: compute the actual raw buyer assets and fee for the candidate units, then decrement units until `rawBuyerAssets + fee <= remainingBudget`.

Resolution

Tenor Team: The issue was resolved in PR#433.

L-13 | Griefing V1=>V2 Migration

Category	Severity	Location	Status
Gas Griefing	Low	src/callbacks/MorphoV1ToV2BorrowCallback.sol	Acknowledged

Description

`MorphoV1ToV2BorrowCallback.onSell()` migrates a borrower from Morpho Blue V1 to Midnight V2 by using `sellerAssets - fee` as the repayment budget for the V1 debt. During callback execution it reads the live V1 debt with `MORPHO_BLUE.expectedBorrowAssets(...)`, then reverts if the fixed repayment budget is greater than that live debt. Because Morpho Blue allows anyone to repay on behalf of a borrower, a third party can front-run a quoted migration with a dust repay. If the migration was quoted as an exact or near-exact full migration, the dust repay lowers the live debt while the signed offer still produces the original `repayBudget`. The callback then reverts with `ExcessRepayment`, forcing the borrower or keeper to re-quote and resubmit the migration.

This is not a direct value-stealing issue: the attacker pays down the borrower's debt and the migration can be re-quoted. It is nevertheless a cheap liveness grief against time-sensitive V1->V2 borrow renewals. Morpho Blue explicitly documents the underlying behavior: a small repay can front-run a repay and make it revert.

Recommendation

Avoid reverting when `repayBudget` is slightly above the live V1 debt. If `repayBudget >= v1Debt`, treat the fill as a final migration: repay by `pos.borrowShares`, migrate all V1 collateral, and base accounting/events on the actual live debt repaid. If surplus loan tokens can remain in the callback after the exact share repay, refund them or ensure they cannot be stranded or reused across users.

Resolution

Tenor Team: Acknowledged.

L-14 | Router Fee Metadata Under-reports Fees

Category	Severity	Location	Status
Unexpected Behavior	Low	src/periphery/resolvers/CallbackFeeAdjuster.sol	Acknowledged

Description

`TakeRouter` delegates fee sizing and post-dispatch accounting to `ICallbackFeeAdjuster` using `actions[i].feeAdjusterData`. The canonical `CallbackFeeAdjuster` decodes that data as `(feeRate, FeeFormula)` and uses the selected formula in both `beforeDispatch()` and `afterDispatch()`. Neither the router nor the adjuster checks that the selected formula matches the callback encoded in the action.

For callbacks that charge a flat percentage fee, such as V2-to-V1 borrow/lend paths, a caller can label the action as `FeeFormula.INTEREST`. Near par, the interest-based formula reports zero or near-zero fee, while the actual callback still charges `percentageFee(...)`. The router can therefore report totals and enforce `maxFill` against a lower fee than the callback-side cost the taker experiences.

This is not arbitrary third-party theft by itself: the transaction still must be submitted by the taker or by an operator that passes the router's authorization check. The protection failure is that router-level limits are only as correct as untrusted per-action metadata.

Recommendation

Do not accept the fee formula as arbitrary caller metadata. Derive the expected formula from the callback address, or validate `(callback, FeeFormula)` pairs before dispatch. Revert if a percentage-fee callback is routed with `FeeFormula.INTEREST`, or if an interest-fee callback is routed with `FeeFormula.PERCENTAGE`.

Resolution

Tenor Team: Acknowledged.

L-15 | Bidirectional Intents Enable Fee-draining Churn

Category	Severity	Location	Status
Gaming	Low	RenewalIntentRatifier.sol	Acknowledged

Description

`RenewalIntentRatifier` stores permissions independently per `(user, callback, sourceMarketId, targetMarketId)` and never consumes them on use. That lets opposite directions remain live at the same time. A realistic lifecycle is:

1. the user enables `V1->V2` to move from V1 fixed-rate exposure into V2 variable-rate exposure
2. a keeper executes that migration
3. later, market conditions flip, so the user enables `V2->V1` to move back into V1
4. a keeper executes that exit
5. the old `V1->V2` intent is still live, so the keeper can immediately move the user back into V2

The protocol never enforces that:

• only one direction may be active

• executing one direction invalidates the reverse direction

• a position cannot be renewed straight back into the market it just exited

Each extra round-trip charges real fees:

• `V1->V2 lend` withdraws `buyerAssets + fee` from the user's vault position

• `V2->V1 lend` deposits only `sellerAssets - fee` back into the vault

The problem is worse on `V2->V1`, where `BaseRenewalRatifier` excludes the flat callback fee from rate validation (reported I-03).

In effect, the `V2->V1` protocol fee is paid out of the user's ratified price budget. The keeper only needs a quote that passes the pre-fee check; the callback can then worsen the user's realised outcome by the omitted flat fee while still passing ratification.

Assuming executable offer liquidity exists, including self-supplied or coordinated liquidity, stale bidirectional intents become a repeatable value-draining churn loop. `V2->V1` becomes valid when `block.timestamp == sourceMaturity - renewalWindow`, and the stale `V1->V2` intent back into that same market is still valid because `targetMaturity == block.timestamp + minDuration`. At that boundary a keeper can immediately alternate the two directions against the same V2 market and stack fees each time.

Recommendation

Invalidate stale directional intent when a migration executes. Consider:

• consuming the executed intent so it cannot be replayed later

• adding nonces or one-shot intent IDs

• requiring explicit re-authorization before a position can migrate back into the market it just exited

Resolution

Tenor Team: Acknowledged.

I-01 | V1->V2 Borrow Renewal Can Spend Target Credit

Category	Severity	Location	Status
Validation	Info	MorphoV1ToV2BorrowCallback.sol	Acknowledged

Description

`MorphoV1ToV2BorrowCallback` assumes the traded `obligationUnits` become new debt on the target Midnight obligation. That is not guaranteed in the actual renewal flow. `RenewalIntentTakeOnBehalf.take()` calls `Midnight.take()` with the borrower as the sell-side taker of a lender `buy` offer. In that branch, Midnight first applies `units` against `sellerPos.credit` before this callback executes:

```
    uint256 sellerCreditDecrease = UtilsLib.min(units, sellerPos.credit);
    uint256 sellerDebtIncrease = units - sellerCreditDecrease;
    uint128 buyerPendingFeeIncrease =
    ...
    sellerPos.credit -= UtilsLib.toUint128(sellerCreditDecrease);
    sellerPos.debt += UtilsLib.toUint128(sellerDebtIncrease);
```

Therefore any credit the borrower already holds on the target obligation is sold first, and only the remainder becomes new debt.

The callback does not guard against that condition. It still uses the received loan tokens to repay the V1 debt and migrates V1 collateral into the target obligation. A keeper can therefore turn a "borrow migration" into a discounted sale of the user's existing target credit whenever the user already lends in the chosen target market. The lender side receives that credit at the renewal price, while the user loses a position that was not covered by their borrow-renewal guardrails. `reduceOnly` does not help here, since the external offer maker is the lender and the borrower is the taker.

Recommendation

Reject V1-to-V2 borrow renewals when the user already has credit on the target obligation. This check needs to happen before `Midnight.take()` mutates the position, since the callback only sees post-trade state and cannot detect credit that was fully consumed in the same call. The ratifier or orchestrator should read `updatePositionView` or `creditOf` for the target obligation and revert if it is non-zero for this renewal type. If that behavior is intentional, document it explicitly and treat target-credit sales as a separate flow.

Resolution

Tenor Team: Acknowledged.

I-02 | Stale Grace Timer Survives Healthy Recovery

Category	Severity	Location	Status
Unexpected Behavior	Info	DelayedLiquidationGate.sol	Acknowledged

Description

`startGracePeriod()` stores a single timestamp per borrower and obligation and `_requireLiquidationAllowed()` later checks only whether the current time still falls inside the window derived from that timestamp. The gate never invalidates `gracePeriodInfo` after the borrower returns healthy and it does not distinguish between a borrower who remained continuously unhealthy and a borrower who recovered before becoming unhealthy again. As a result, a borrower can become unhealthy, have anyone arm the grace timer, cure the position and then become unhealthy again before the original liquidation window expires. The second unhealthy episode inherits the old timer and can be liquidated immediately, even though the documentation describes a fresh `unhealthy -> startGracePeriod -> grace period -> liquidate` lifecycle. This does not let liquidators seize a healthy position, because Midnight still re-checks current health at liquidation time. The bug is narrower and more precise: a borrower who recovers loses the promised grace period on the next unhealthy episode.

Recommendation

Make grace-period validity track continuous unhealthy state, not just elapsed wall-clock time since some historical dip. Move the "unhealthy since" tracking into Midnight, where collateral and debt updates can invalidate stale timers whenever an account becomes healthy again. If the gate must keep the state, add an explicit invalidation path tied to every action that can restore health and require a fresh `startGracePeriod()` after recovery.

Resolution

Tenor Team: Acknowledged.

I-03 | V2-to-V1 Fees Bypass Rate Limits

Category	Severity	Location	Status
Validation	Info	Constants.sol	Acknowledged

Description

`MAX_FEE_RATE_V2_V1` permits non-zero fees on V2->V1 renewals. That is unsafe because `BaseRenewalRatifier._feeRateForRateCheck()` returns zero for both V2->V1 callbacks before the ratifier builds the price checked by `PriceLib.satisfiesRateLimit()`. The callbacks still charge `CallbackLib.percentageFee()` on principal. Consequently, the fee is not included in the user's `limitRatePerSecond` protection.

An offer that sits exactly on the user's rate boundary can pass ratification and then settle worse once the fee is added or deducted. This becomes severe near maturity. `_computeDuration()` uses `sourceMaturity - block.timestamp` for V2->V1 exits, so the same 25 bps principal fee is roughly equivalent to 91% APR with 1 day remaining and about 131,400% APR with 1 minute remaining. After maturity, the rate model collapses to par because duration is zero, but the fee is still collected. Because `setFeeConfig()` is owner-controlled and takes effect immediately, a malicious or compromised fee admin can raise V2->V1 fees up to this cap and extract value from pending renewals without tripping `InvalidOfferRate`.

Recommendation

Do not include the V2-to-V1 flat percentage fee in the interest-rate check if the intended policy model is that V2-to-V1 exits are checked pre-fee. Instead, document that distinction directly in the ratifier code and deployment docs.

The documentation should make clear that, for V2-to-V1 renewals, `limitRatePerSecond` constrains only the pre-fee exchange rate. The flat exit fee is controlled separately by `MAX_FEE_RATE_V2_V1`, market/action fee configuration and the ratifier owner trust model.

Resolution

Tenor Team: The issue was resolved in PR#441.

I-04 | Sell Takes Can Spend Router-held Tokens

Category	Severity	Location	Status
Warning	Info	TakeRouter.sol	Acknowledged

Description

`_dispatchMidnightTake()` treats a sell offer with `takerCallback == address(0)` as a case where the router must approve Midnight to pull loan tokens. This matches `Midnight.take()`'s payer selection, but it means the payment source is the router contract itself, not `params.taker`.

An attacker can self-authorize the router on Midnight, set themselves as `params.taker`, and submit a `MIDNIGHT_TAKE` against an attacker-controlled sell offer. If the router holds the obligation's `loanToken`, Midnight pulls `buyerAssets` from the router, sends `sellerAssets` to the maker-side receiver, and credits the purchased position to the attacker-controlled taker. The router never checks that the spent loan tokens came from the taker who passed the authorization check.

Consequently any loan-token balance stranded on the public router is globally spendable through this direct sell execution. The code and tests intentionally support router-funded settlement for this case, but the public `TakeRouter` is not documented as a custody component and has no per-user balance accounting or sweep function.

The same inherited logic exists on `TenorAdapter`, but the adapter impact is weaker. `TenorAdapter` inherits `CoreAdapter.erc20Transfer`, so a user can append `erc20Transfer(loanToken, user, type(uint256).max)` to the same Bundler3 multicall and sweep the adapter's full residual token balance. If the user fails to do that, the stale adapter balance is already publicly sweepable through `CoreAdapter.erc20Transfer`, without needing this router sell execution. Therefore the adapter residual case does not justify high severity by itself.

Recommendation

Do not let the public router become the payer for direct sell takes. The smallest safe fix is to reject `MIDNIGHT_TAKE` actions where `!action.offer.buy && d.takerCallback == address(0)`. If router-funded settlement is intentionally needed, gate it behind a separate trusted mode and account for the exact balance supplied for the current action. Do not fund it from the router's aggregate token balance. Adapter integrations should also append an explicit final sweep when they transfer tokens into `TenorAdapter`.

Resolution

Tenor Team: The issue was resolved in PR#401.

I-05 | Supply Collateral Clamp Quotes Unhealthy Fills

Category	Severity	Location	Status
Warning	Info	SupplyCollateralCallbackClamp.sol	Acknowledged

Description

`SupplyCollateralCallbackClamp._maxUnitsFromDebtLimit()` computes an initial maximum fill from a linearized health formula, then simulates the callback collateral for that fill. If the simulated position is unhealthy, it performs one correction:

```
    if (forwardLimit < forwardDebt) {
    maxUnits = forwardLimit > currentDebt ? forwardLimit - currentDebt : 0;
    }
```

That correction is not self-consistent. `forwardLimit` was computed using the old `maxUnits`, but the callback collateral is proportional to `sellerAssets`. When `maxUnits` is reduced, `sellerAssets` can also shrink. The callback then supplies less collateral than the value used to compute the corrected limit.

Consequently the clamp can return a positive fill that still makes `Midnight.take()` revert with `SellerIsLiquidatable`, even though a smaller fill would succeed.

Recommendation

Make the health and max-LTV cap converge before returning a fill size. The simplest fix is to replace the single correction step with a monotone binary search over units, using the same seller-asset conversion and pro-rata collateral calculation that execution will use.

Alternatively, repeat the forward simulation after each correction until the returned `maxUnits` satisfies `forwardLimit >= currentDebt + maxUnits`. The implementation should also keep the existing conservative rounding behavior so the clamp may return slightly less than the true maximum, but never more than an executable fill.

Resolution

Tenor Team: Acknowledged.

I-06 | TENOR_VAULT_V2 Clamp Reads Wrong Data

Category	Severity	Location	Status
Unexpected Behavior	Info	V1ToV2LendClamp.sol	Resolved

Description

`V1ToV2LendClamp.clamp` always passes `offer.callbackData` into `_ownerWithdrawable`. The TENOR_VAULT_V2 branch then reads `morphoBlueMarketId` from that byte array with assembly. That is wrong for the normal V1-to-V2 lend renewal shape. The lender is the taker and the lend callback is passed as `takerCallback`, so the real `MorphoV1ToV2LendCallback.CallbackData` lives in `TakeOnBehalfData.takerCallbackData`. `offer.callbackData` belongs to the maker's seller callback and is commonly empty because the seller offer does not need a callback.

The result is that the TENOR_VAULT_V2 clamp cannot see the market ID that the callback will use. With empty `offer.callbackData`, it reads `bytes32(0)`, queries the zero Morpho Blue market, and returns zero liquidity even when the lender has vault shares and the real Morpho Blue market is liquid. The audit test `test_tenorVaultV2ClampCannotReadTakerCallbackDataOnSellOffer()` shows this directly: the clamp returns zero for the real SELL-offer shape, then returns a positive cap only after the same callback data is redundantly copied into the otherwise unused `offer.callbackData`.

```
This blocks TENOR_VAULT_V2 clamped V1-to-V2 lend renewals unless the seller signs unused callback data solely for the clamp. If integrators fall back to the VAULT_V2 or generic ERC4626 branch to avoid the zero cap, they lose the intended Morpho Blue liquidity bound and can produce revert-only fills when the underlying market is illiquid.
```

Recommendation

Do not read TENOR_VAULT_V2 liquidity data from `offer.callbackData` for taker-callback renewals. Move `morphoBlueMarketId` into `V1ToV2LendClampData`, or extend the router/clamp interface so the clamp receives the actual taker callback data used by `Midnight.take()`.

The clamp should also ABI-decode the callback data or clamp data instead of using an unchecked assembly load. That makes malformed data fail clearly and allows the clamp to validate that the vault, fee rate, and market ID being capped match the callback execution.

Resolution

Tenor Team: The issue was resolved in PR#414.

I-07 | Vault Supply Clamp Uses An Unbound Tick

Category	Severity	Location	Status
Unexpected Behavior	Info	VaultSupplyClamp.sol	Resolved

Description

`VaultSupplyClamp.clamp` computes the seller top-up cap from `offer.tick`, but `MorphoV2SupplyVaultSharesCallback.onSell` computes the actual top-up from `callbackData.tick`. The interface says those ticks must match, but neither the callback nor the clamp enforces that relationship. The clamp also reads only the third ABI word from `offer.callbackData`, so it does not reject malformed or inconsistent remaining callback data before returning a cap.

For sell offers, a maker can sign an offer with a lower `offer.tick` and a higher `callbackData.tick`. The clamp then reports a fill size based on the smaller top-up. When the take executes, the callback pulls the larger top-up and can revert because the seller's balance or allowance is insufficient for the amount that the clamp did not size.

Consequently, routed batch or keeper can select a fill that the clamp advertises as safe and then lose the action to a callback revert.

Recommendation

Bind the callback tick to the offer tick before returning a nonzero clamp value. The clamp should ABI-decode the full `CallbackData`, require the expected length and structure and require `callbackData.tick == offer.tick`.

Resolution

Tenor Team: The issue was resolved in PR#411.

I-08 | Vault Supply Clamp Omits Solvency Bound

Category	Severity	Location	Status
Validation	Info	VaultSupplyClamp.sol	Resolved

Description

`VaultSupplyClamp.clamp` returns a fill cap from the seller's top-up balance and allowance only. It does not check whether the shares minted by the callback will make the seller healthy under the obligation oracle and LLTV. The comment justifies this by assuming the vault-to-loan-token oracle price only increases. That assumption does not hold for the intended VaultV2 collateral. The project documentation states that underlying bad debt can reduce VaultV2 `totalAssets` and drop the share price. If the share price or oracle value decreases, or if `additionalDepositPercent` is below the current LLTV requirement, the clamp can return a positive unit amount that will always revert at Midnight's final `SellerIsLiquidatable` check.

This does not create bad debt because Midnight still enforces health after `onSell`. The issue is liveness and routing reliability. A router or keeper using the clamp can select a fill that the clamp reports as available, only to revert after the callback deposits and supplies shares.

Recommendation

Make solvency part of the clamp's fill bound. For vault-share collateral, cap units by the seller's post-fill health using the current oracle price, LLTV, expected share output, existing debt, existing credit and existing collateral. If the clamp intentionally avoids oracle reads, document that it is only a balance/allowance clamp and must be paired with an independent health-aware bound before routed execution.

Resolution

Tenor Team: The issue was resolved in PR#415.

I-09 | Grace Period Skipped Once Obligation Matures

Category	Severity	Location	Status
Validation	Info	src/gates/DelayedLiquidationGate.sol	Resolved

Description

`DelayedLiquidationGate` enforces the grace period, liquidation window, and the exclusive priority-liquidator sub-window only while the obligation is pre-maturity. Once `block.timestamp` crosses `obligation.maturity`, the gate's checks are skipped entirely and any liquidator can proceed — regardless of whether the grace period has elapsed or whether a priority liquidator was assigned.

There is no validation at `startGracePeriod` that the configured `GRACE_PERIOD + LIQUIDATION_PERIOD` fits inside the remaining time to maturity. If a position becomes unhealthy close to maturity, both the borrower's grace period (intended as a self-cure window) and the priority liquidator's exclusive sub-window can collapse — partially or entirely — the moment maturity is reached.

Recommendation

Either:

1. Validate at `startGracePeriod` that `obligation.maturity > block.timestamp + GRACE_PERIOD + LIQUIDATION_PERIOD`, reverting otherwise; or

2. Explicitly document that the grace period and priority window are best-effort and are truncated or skipped once the obligation matures.

Resolution

Tenor Team: The issue was resolved in PR#412.

I-10 | Factory Allows Near-total Oracle Deviation

Category	Severity	Location	Status
Validation	Info	OracleWithValidationCheckFactory.sol	Acknowledged

Description

`createOracleWithValidationCheck()` only rejects `maxOracleDeviation >= 1e18`. Therefore the factory accepts values arbitrarily close to 100%. That bound is enough to prevent an exact 100% deviation, but it is not a meaningful safety bound for a lending oracle. With `maxOracleDeviation = 1e18 - 1`, a primary price of `1e36` accepts a validation price of `1e18`. The wrapper is still factory-deployed and `isDeployedOracle` is true, but the validation oracle can be almost zero relative to the primary price.

This matters because Midnight computes collateral debt capacity from the returned primary price. A factory-valid wrapper can therefore be configured so the secondary oracle provides almost no protection before the price is used in borrowing-limit and liquidation checks. This is a configuration-boundary issue rather than a permission bypass. It requires a market or deployment process to accept an unsafe deviation value.

Recommendation

Do not treat `maxOracleDeviation < 1e18` as a sufficient risk bound. Add deployment tooling or governance checks that enforce a market-specific maximum deviation before an oracle can be used as collateral.

For lending markets, derive the maximum from the collateral's LLTV and liquidation assumptions. If the current primary-denominated formula remains, use the stricter bound documented in the existing deviation finding rather than comparing `maxOracleDeviation` directly to `1 - LLTV`.

Resolution

Tenor Team: Acknowledged.

I-11 | Zero-owner Oracle Docs Cannot Deploy

Category	Severity	Location	Status
Documentation	Info	README.md	Resolved

Description

The oracle README says to set `owner` to `address(0)` to make validation immutable. That deployment cannot succeed with the current dependency tree. `OracleWithValidationCheck` passes `initialOwner` into OpenZeppelin `Ownable`, and `Ownable` reverts when the initial owner is `address(0)`.

Consequently the documented immutable deployment method is not available through either the factory or a direct constructor call. The actual way to remove owner powers is to deploy with a nonzero owner and then call `renounceOwnership()`. That sequence has different safety requirements because the owner can pause validation before renouncing, which leaves the oracle permanently primary-only.

Operators following the docs will either ship a failing transaction or use a renounce sequence without the explicit pre-renounce state checks that this contract needs.

Recommendation

Update the README to describe the real immutable deployment procedure. The safe runbook is to deploy with a nonzero owner, verify that `validationCheckPaused == false`, verify that both oracles return acceptable prices, and only then renounce ownership if immutability is desired.

Resolution

Tenor Team: The issue was resolved in PR#changes.

I-12 | Lender Rate Floors Bypassed After Early Repay

Category	Severity	Location	Status
Logical Error	Info	BaseRenewalRatifier.sol, MorphoV2ToV2LendWithdrawable.sol	Acknowledged

Description

`BaseRenewalRatifier` is supposed to stop a permissionless keeper from renewing a lender into a bond priced below the lender's configured floor rate. For `LEND_V2_TO_V2_WITHDRAWABLE`, that protection is computed with the wrong duration once the source bond has already been repaid early.

The ratifier always prices V2->V2 renewals using `targetMaturity - sourceMaturity`. That assumption is valid for the borrower-side renew path, where the source debt remains live until it is repaid inside the callback. It is not valid for the lender-side `withdrawable` path. Once the borrower of the source obligation repays early, the lender's capital is no longer locked until `sourceMaturity`; it is immediately redeployable. The callback then pulls that cash out of the source obligation at execution time and uses it to fund the new target bond. So after an early repay, the economic tenor of the renewed lend is `targetMaturity - block.timestamp`, not `targetMaturity - sourceMaturity`.

This direction matters because `LEND_V2_TO_V2_WITHDRAWABLE` is treated as `userIsBuy = true`, so the rate check enforces a lender floor. In `PriceLib`, a shorter duration produces a higher zero-coupon price. That makes the lender floor easier to satisfy, which lets the taker pay too much for the target bond and accept a lower yield than their configured minimum. In practice, the borrower on the other side of the target obligation receives financing at a better rate than the lender allowed, defeating the ratifier's primary purpose.

This is exploitable under normal operation, for example:

• lender originally lends `100` until `t + 4w`

• source borrower fully repays at `t + 1w`

• renewal executes immediately at `t + 1w` into a target maturing at `t + 5w`

• ratifier duration: `t+5w - t+4w = 1w`

• real deployed duration: `t+5w - t+1w = 4w`

At a 5% floor, the 1-week price floor is materially higher than the 4-week price floor. A new borrower can therefore borrow against that lender's liquidity at a lower yield than the lender approved, simply by waiting for early repayment to create withdrawable source funds and then matching the renewal against a target offer priced at the inflated floor.

Recommendation

For `LEND_V2_TO_V2_WITHDRAWABLE`, compute duration from execution time, not from the old maturity: `targetMaturity - block.timestamp`

Resolution

Tenor Team: Acknowledged.

I-13 | Withdrawals Escape Lazy Bad Debt

Category	Severity	Location	Status
Validation	Info	Midnight.sol	Acknowledged

Description

Bad debt is socialized only when `liquidate()` is called and the bad-debt branch updates the obligation's `lossIndex`. Until that happens, `withdraw()` still lets any lender burn current credit and withdraw from the global `withdrawable` bucket at par.

An informed lender can therefore exit before losses are realized. The lender withdraws their credit against existing `withdrawable` liquidity, then the same account or another liquidator calls `liquidate()` on an already-bad borrower. The subsequent `lossIndex` update slashes only the lenders who still hold credit. The exiting lender avoided their share of the loss because their credit was already burned at par.

This creates a bank-run dynamic: repayments are paid out first-come-first-served, while losses are recognized lazily. `DelayedLiquidationGate` can make the timing worse because it may block liquidation during a grace or priority period while withdrawals from the same obligation remain open.

Recommendation

Do not allow par withdrawals to outrun known bad debt. Possible fixes include realizing pending bad debt before withdrawals, freezing or haircutting withdrawals while liquidation grace periods are active, or moving to indexed repayment/loss accounting where each credit holder's withdrawal entitlement and loss exposure are applied consistently over time.

Resolution

Tenor Team: Acknowledged.

I-14 | Renewal Skips Collateral Missing From Target

Category	Severity	Location	Status
Unexpected Behavior	Info	CollateralTransferLib.sol	Resolved

Description

`transferCollaterals` walks every collateral token in the source obligation, but only transfers a token when `targetObligation.findCollateral()` returns `found`. If the target obligation does not list that source token, the helper leaves `collateralAmounts[i]` at zero and keeps going.

This is safe for a generic helper only if the caller has already opted into partial collateral migration. `MorphoV2ToV2BorrowRepay.onSell` does not enforce that. It only checks that the source and target obligations use the same `loanToken` before it repays source debt and calls `transferCollaterals`. The ratifier binds the exact source and target market IDs, but it does not prove that the two obligations have the same collateral token set. The V2-to-V2 borrow clamp documents the opposite assumption: source and target obligations should be identical except for maturity.

Consequently, a V2-to-V2 borrow renewal can repay some or all source debt, emit `PositionRenewed` and silently leave source collateral behind. On a final fill, the source debt can be fully closed while unmatched collateral remains in the old obligation. On a partial fill, the borrower can end with target debt plus residual source collateral state that the surrounding renewal machinery treats as migrated.

Recommendation

Enforce the renewal-specific compatibility requirement before calling `transferCollaterals`. For the documented V2-to-V2 borrow renewal, require the source and target obligations to have the same collateral token set and compatible collateral parameters, not just the same `loanToken`. If partial collateral migration is intentional, make it explicit. The callback should reject unapproved mismatches, emit clear residual-state information for skipped tokens and the README, clamp comments and source-closure properties should be updated to describe that behavior.

Resolution

Tenor Team: The issue was resolved in PR#417.

I-15 | TakeRouter Returns Adjusted Totals Only

Category	Severity	Location	Status
Unexpected Behavior	Info	TakeRouter.sol	Acknowledged

Description

`TakeRouter` tracks two different aggregate result sets during batch execution: `rawTotals`, which match the raw fill amounts that Midnight writes into `consumed[maker][group]`, and `totals`, which are post-adjustment values after `feeAdjuster.afterDispatch` tilts the result against the taker. This distinction is documented inline in `_execute`.

However, the external `execute` entrypoint returns only `totals`, and `BatchExecuted` also emits only the adjusted totals. An integrator that assumes these returned values are equivalent to Midnight's raw fill accounting can drift from `consumed` when callback fee adjustment is enabled.

```
    function execute(ExecuteParams calldata params, Action[] calldata actions)
    external
    virtual
    ...
    rawTotals[RouterLib.FILL_BUYER_ASSETS] += buyerAssets;
    rawTotals[RouterLib.FILL_SELLER_ASSETS] += sellerAssets;
```

Recommendation

If external consumers are expected to reconcile against Midnight `consumed` accounting, expose `rawTotals` in the `execute` return values or emit them in a separate event. Otherwise, document more explicitly that `execute` and `BatchExecuted` expose taker-facing adjusted totals only and must not be used as a proxy for raw Midnight fill accounting.

Resolution

Tenor Team: Acknowledged.

I-16 | V2-to-V2 Lend Callback Self-funds Fresh Credit

Category	Severity	Location	Status
Unexpected Behavior	Info	MorphoV2ToV2LendWithdrawable.sol	Acknowledged

Description

`MorphoV2ToV2LendWithdrawable.onBuy` trusts `callbackData.sourceObligation` as the obligation from which the buyer already has withdrawable credit. It only checks that the source and target loan tokens match. It does not reject the case where the source obligation is the same obligation currently being bought into.

This is a follow-up manifestation of `H-03: Fresh credit drains old repayments`. In `Midnight.take`, the buyer receives fresh credit before `onBuy` executes and before Midnight pulls payment from the buyer callback. Therefore a raw `Midnight.take` or a router `MIDNIGHT_TAKE` action can pass `sourceObligation == obligation` to this public callback. In those paths, the taker supplies `takerCallbackData`, which is decoded by this callback as `CallbackData`; that makes `sourceObligation`, `feeRate`, `feeRecipient` and `tick` attacker-controlled for the exploit path. The callback then calls `MORPHO_MIDNIGHT.withdraw(callbackData.sourceObligation, neededAssets, buyer, address(this))` and burns the buyer's just-minted credit against the same obligation's old `withdrawable` pool. Because the raw-take path lets the attacker choose the callback data, the attacker can set `feeRate == WAD` and `feeRecipient == attacker`. With those values, the callback withdraws approximately the full obligation unit amount from old withdrawable liquidity. It sends the discount component to the attacker as a callback fee and approves only `buyerAssets` back to Midnight so settlement can complete. The attacker starts with no loan tokens, the borrower receives the discounted take price and old withdrawable liquidity is drained to fund the new borrower debt.

The canonical renewal ratifier constrains `feeRate` and `feeRecipient` only when the sanctioned renewal-intent path is used and the `V2ToV2LendWithdrawableClamp` rejects or zeroes self-renewal configurations only when that optional clamp is used. Neither policy layer protects raw Midnight offers or `TakeRouter` actions that use `ActionType.MIDNIGHT_TAKE`, because this callback is callable by Midnight directly with caller-provided callback data.

This is repeatable but not unbounded. Each successful execution consumes the target obligation's existing `withdrawable` liquidity one-for-one, so the path stops for that obligation once the old withdrawable pool is empty unless later repayments refill it. The attacker needs both old withdrawable liquidity to drain and a way to mint fresh credit in the same obligation.

There are two practical cases:

1. If honest borrower sell offers already exist, the attacker can take those offers with the public callback. The honest borrower receives the discounted loan proceeds, the attacker captures the discount as the callback fee, and pre-existing withdrawable lenders are left funding the new borrower debt.
2. If no suitable honest borrower offer exists, the attacker can use a second address as the borrower because `Midnight.take` only rejects exact same-address self-takes. In that case, the attacker-controlled borrower must still provide acceptable collateral and ends up with the new debt. The path can still drain old withdrawable liquidity, but profitability then depends on the borrower's collateral, liquidation/default assumptions, and the cost of creating that debt.

Recommendation

Reject same-obligation source and target use inside the callback itself. The guard should be local to `MorphoV2ToV2LendWithdrawable.onBuy` because external ratifiers, clamps and routers are optional policy layers.

```
    bytes32 targetObligationId = IdLib.toId(obligation, block.chainid, address(MORPHO_MIDNIGHT));
    bytes32 sourceObligationId = IdLib.toId(callbackData ... Obligation, block.chainid,
address(MORPHO_MIDNIGHT));
    if (sourceObligationId == targetObligationId) revert SelfRenewal();
```

Also enforce any intended maturity and fee constraints in the callback when direct `Midnight.take` usage is supported. At minimum, reject `callbackData.sourceObligation.maturity >= obligation.maturity`, reject positive fees with a zero fee recipient, and keep the callback tick bound to the offer tick through the caller's data model.

Resolution

Tenor Team: Acknowledged.

I-17 | Renewal Grants Survive Midnight Revocation

Category	Severity	Location	Status
Trust Assumptions	Info	RenewalIntentTakeOnBehalf.sol, RenewalIntentRatifier.sol	Acknowledged

Description

`RenewalIntentTakeOnBehalf.setIsAuthorized()` and `RenewalIntentRatifier.setParams()` accept calls from any account that is authorized by the user on Midnight at the time of the write. The resulting renewal-layer state is stored locally and is not linked to that Midnight authorization afterward.

Consequently, a temporarily authorized delegate can plant a ratifier authorization and permissive renewal params, then keep that renewal surface alive after the user revokes the delegate on Midnight. The revoked delegate can no longer clean up the state it wrote, but the planted ratifier authorization still passes `RenewalIntentTakeOnBehalf.take()` because execution only checks `isAuthorized[intent.user][intent.ratifier]`.

Recommendation

Do not let generic Midnight delegates create persistent renewal-layer authority unless that is explicitly part of the trust model. Require `msg.sender == onBehalf` for `setParams()` and `setIsAuthorized()`.

Resolution

Tenor Team: Acknowledged.

I-18 | V2-to-V1 Borrow Exits Lack Target Health Buffer

Category	Severity	Location	Status
Validation	Info	MorphoV2ToV1BorrowCallback.sol	Acknowledged

Description

`MorphoV2ToV1BorrowCallback.onBuy()` only checks that the target Morpho Blue market uses the same loan token as the source obligation and that the target collateral token appears somewhere in the source obligation. It does not require the target market to use the same oracle or LLTV as the source collateral. It also does not require any target-side health buffer before creating the Morpho Blue borrow.

This matters because `V2ToV1BorrowClamp` assumes the source and target have the same loan token, collateral token, LLTV, and oracle. The clamp does not check target health; it only caps by source debt and target market liquidity. Therefore a configured migration can pass the local callback checks while entering a target market with stricter LLTV or different oracle assumptions. The migration may revert late at Morpho Blue's binary health check or succeed with little liquidation margin under the target market.

The borrower chooses or authorizes the target tuple, so this is primarily a migration safety and product-correctness issue. The code and clamp comments promise a stronger compatibility model than the callback enforces.

Recommendation

Require the selected target market to match the source collateral's oracle and LLTV unless the borrower has explicitly opted into a different target risk profile. Add a target-side health-buffer check that accounts for fees, existing target debt and the actual collateral amount moved.

Resolution

Tenor Team: Acknowledged.

I-19 | Stale Documentation

Category	Severity	Location	Status
Documentation	Info	GLOBAL	Resolved

Description

The repository documentation and contain several stale or contradictory statements relative to the current implementation. None of these are standalone runtime vulnerabilities, but they can mislead auditors, integrators, and offchain tooling that use the docs as the source of truth:

• `DIMENSIONAL_UNITS.md:13` says higher tick means lower price / higher discount. In reality, `TickLib.tickToPrice()` is monotone increasing, so higher tick means higher price / lower discount.

• `DIMENSIONAL_UNITS.md:17` states the price domain is `(0, WAD]`. In reality, `tickToPrice(0) == 0`, so the implementation domain includes zero: `[0, WAD]`.

• `DIMENSIONAL_UNITS.md:48` says tick prices round to `1e13` multiples. In reality, `tickToPrice()` rounds with `.divHalfDownUnchecked(5e12) * 5e12`.

• `PROPERTIES.md:34` refers to `MAX_TICK (990)`. In reality, `MAX_TICK = 1046`.

• `PROPERTIES.md:39` says `tickToPrice(0) > 0`. In reality, tests assert `tickToPrice(0) == 0`.

• `docs/audit-checklist.md:59` says `tick 990 = near par`. In reality, par is reached at `tickToPrice(1046) == 1e18`.

• `docs/audit-checklist.md:117` says the effective-price fee model uses all rounding down on both seller and buyer sides. In reality, `CallbackLib` uses mixed rounding: seller-side effective price / budget uses `mulDivUp`, while buyer-side effective price / budget uses `mulDivDown`.

• `CallbackData.feeRate` in V2 to V1 interfaces: `Fee rate on sellerAssets in WAD (max 1%, e.g., 0.01e18 = 1%)` but declared constant `MAX_FEE_RATE_V2_V1 = 0.0025e18`

• Several callback interfaces say `feeRecipient address(0) = no fee`; runtime callbacks do not use `feeRecipient` as the no-fee sentinel. Fees are disabled by `feeRate == 0` or by a computed zero fee.

• IMorphoV2SupplyVaultSharesCallback first states the vault top-up is `ceil(sellerAssets * additionalDepositPercent / WAD)`, but the implementation computes `ceil(obligationUnits ceil(tickToPrice(tick)  additionalDepositPercent / WAD) / WAD)`

Recommendation

Fix the documentation contradictions.

Resolution

Tenor Team: The issue was resolved in PR#437.

I-20 | Sentinel Silently No-ops Takes

Category	Severity	Location	Status
Logical Error	Info	src/bundler/TakeRouterAdapterBase.sol :77-83	Resolved

Description

`TakeRouterAdapterBase._executeResolvingSentinels` overloads `type(uint256).max` on `ExecuteParams.maxFill` / `minFill` as a request to derive the cap from on-chain state. For `fillIndex == FILL_OBLIGATION_UNITS`, `_resolveSentinel` returns `debtOf(id, taker)`, falling through to `creditAfterUpdate` if debt is zero, and returning `0` if neither exists. The intended semantic is "fill up to my existing position size" — meaningful only for renewal or close-out flows where the taker already has a position in the obligation.

The `type(uint256).max` value, however, is the canonical ERC-20 idiom for "unlimited / no cap." An integrator unfamiliar with the renewal-only semantic — or wired by a frontend reusing the ERC-20 convention — passes `maxFill = type(uint256).max` for an opening take (no prior debt or credit). `_resolveSentinel` returns `0`, and `_execute`'s loop guard `totals[fillIndex] >= maxFill` fires `0 >= 0` on the first iteration, breaking before any action runs. With the default `minFill = 0`, the post-loop floor check `totals[fillIndex] < minFill` does not revert, so `BatchExecuted` is emitted with all-zero amounts. The caller observes a successful transaction that filled nothing, with valid signatures, valid approvals, and no error to diagnose.

Recommendation

Either (a) revert with a named error (e.g. `SentinelResolvedToZero` / `SentinelRequiresExistingPosition`) when `_resolveSentinel` returns `0` and the caller passed `type(uint256).max` for the corresponding bound — opening takes that misuse the sentinel then fail loudly instead of silently; or (b) document at the `ExecuteParams` site that `maxFill = type(uint256).max` is a renewal/close-out helper, not an "unlimited" sentinel, and direct opening flows to pass concrete values. Option (a) also subsumes the `FILL_BUYER_ASSETS` BUY-offer path in Guardian `69eb590a` with a single guard.

Resolution

Tenor Team: The issue was resolved in PR#436.

I-21 | Stable Max Sentinel Nets Gas

Category	Severity	Location	Status
Unexpected Behavior	Info	src/bundler/TakeRouterAdapterBase.sol	Resolved

Description

On Stable, USDT0 is both the native gas token and an ERC20 token. Gas is precharged from the transaction sender before EVM execution, then refunded after execution. User-called Bundler3 routes that use `type(uint256).max` as a "use my full balance" sentinel therefore resolve against the user's post-gas-reservation USDT0 balance, not the pre-submit wallet balance.

This affects TenorAdapter routes that first transfer the user's max available USDT0 into the adapter, then rely on adapter-balance sentinels or max-balance actions such as `FILL_BUYER_ASSETS`, `midnightRepay(..., assets = type(uint256).max)`, or `midnightSupplyCollateral(..., assets = type(uint256).max)`.

Depending on the route's minimum-fill and exact-amount requirements, the transaction can either succeed with a smaller fill than the user's pre-submit full-balance expectation, or revert because the post-precharge balance is insufficient for the exact downstream action.

Recommendation

For Stable deployments, document that max-balance sentinels mean "post-gas-reservation balance." Frontends and keepers should leave USDT0 gas headroom or use explicit fixed `maxFill` / `minFill` values when exact fills are required.

Resolution

Tenor Team: The issue was resolved in PR#434.

I-22 | Delayed Gate Blocks Gated Vault Liquidations

Category	Severity	Location	Status
DoS	Info	MidnightVaultExecutor.sol	Acknowledged

Description

`MidnightVaultExecutor.liquidateAndRedeem()` cannot liquidate an obligation that uses `DelayedLiquidationGate` as its `liquidatorGate`. This is not just a missing deployment allowlist entry. The executor and the delayed gate use incompatible liquidation call shapes.

The executor calls `MORPHO_MIDNIGHT.liquidate()` directly, so Midnight checks `DelayedLiquidationGate.canLiquidate(address(executor))`. The delayed gate only returns true for `address(this)`, so the executor reverts with `LiquidatorGatedFromLiquidating`.

The alternative is also broken for share-gated VaultV2 collateral. Liquidators are expected to call `DelayedLiquidationGate.liquidate()` directly, but Midnight transfers seized collateral to `msg.sender`, which is then the delayed gate. If the collateral token is a VaultV2 share with `receiveSharesGate` set, that transfer requires `DelayedLiquidationGate` to be allowed by `canReceiveShares`. The documented share-gated setup allowlists `MidnightVaultExecutor`, Midnight and the vault-share callbacks, but not `DelayedLiquidationGate`.

Consequently, the executor cannot pass the delayed liquidation gate, while the delayed gate cannot receive the seized shares. Simply allowlisting `DelayedLiquidationGate` only fixes the second revert. It does not make `MidnightVaultExecutor.liquidateAndRedeem()` usable with delayed-gated obligations and it leaves the delayed gate holding raw VaultV2 shares that its current `onLiquidate()` logic tries to forward to the external liquidator. If that liquidator is not allowlisted to receive shares, the forward transfer can still revert. If arbitrary liquidators are allowlisted, the share-gating design is weakened because protected shares can leave the curated set.

This affects an intended protocol configuration: VaultV2 share collateral is documented as supported behind `VaultV2AllowlistGate`, `liquidateAndRedeem()` is documented as the liquidation mechanism that gives liquidators underlying assets instead of shares and `DelayedLiquidationGate` is documented as a general `liquidatorGate` and liquidation router for Midnight obligations. If these components are wired together, underwater or post-maturity vault-share positions can become unliquidatable and bad debt can remain unresolved.

Recommendation

Make the delayed liquidation and vault-share redemption components compatible instead of relying on deployment guidance alone.

One safe design is to route vault-share liquidations through the delayed gate with the executor as the gate-level liquidator. The executor should pass callback data that lets `DelayedLiquidationGate.onLiquidate()` forward seized shares to the executor, then `MidnightVaultExecutor.onLiquidate()` can redeem those shares and collect repayment from the original liquidator. This still requires `DelayedLiquidationGate` and `MidnightVaultExecutor` to be allowed by the VaultV2 `receiveSharesGate`, because Midnight first transfers seized shares to the gate and the gate then transfers them to the executor.

Alternatively, fold vault-share redemption directly into `DelayedLiquidationGate` or deploy a dedicated delayed vault liquidation router. In either design, the gate should not forward raw gated vault shares to arbitrary liquidators. It should redeem shares into underlying assets before delivery or forward shares only to an allowlisted redemption contract.

Resolution

Tenor Team: The issue was resolved in PR#455.

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
L-01	executeAndConsume Does Not Bind Group Dimension	Validation	Low	Acknowledged
L-02	Unbound Collateral Supply Denominator	Validation	Low	Acknowledged
L-03	Renewal Window Can Underflow Before Validation	Validation	Low	Resolved
L-04	Optional Router Fee Adjustment	Validation	Low	Acknowledged
L-05	Router Limits Assume Homogeneous Actions	Validation	Low	Resolved
L-06	Clamp Data Not Bound To Callback Data	Validation	Low	Acknowledged
L-07	Renewal Can Rescue A Liquidatable Obligation	Trust Assumptions	Low	Acknowledged
I-01	Effective Units Panic On Long Maturities	Validation	Info	Resolved
I-02	Open Liquidator Choice Delays Liquidations	Warning	Info	Acknowledged
I-03	Rate Limits Ignore Settlement Rounding	Warning	Info	Acknowledged
I-04	Standing Renewals Lack Amount Bounds	Warning	Info	Acknowledged
I-05	Vault Withdraw Callback Allows Position Crossing	Warning	Info	Acknowledged
I-06	Stale MAX_FEE_RATE_V2_V1 In Docs	Documentation	Info	Resolved

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
I-07	allowRevert Scope Is Incomplete	Documentation	Info	Acknowledged

L-01 | executeAndConsume Does Not Bind Group Dimension

Category	Severity	Location	Status
Validation	Low	TakeRouterAdapterBase.sol	Acknowledged

Description

`executeAndConsume()` advances the caller's `consumeGroup` by `rawTotals[params.fillIndex]`. Midnight offer groups do not carry denomination metadata. The same `consumed[user][group]` counter can represent units, seller assets or buyer assets depending on whether the signed offer used `maxUnits`, `maxSellerAssets`, or `maxBuyerAssets`. Midnight's `take()` path enforces this distinction from the offer fields, but `executeAndConsume()` only receives a `bytes32 consumeGroup` and the router's `fillIndex`. As a result, the adapter cannot verify that the amount it writes into `consumeGroup` is denominated the same way as the standing offer the caller meant to consume or cancel. If an integration uses `executeAndConsume()` with a unit-capped standing offer but chooses `fillIndex = FILL_BUYER_ASSETS` for the routed fill, the adapter writes buyer assets into a counter that the standing offer later compares against units. The reverse mismatch can also over-consume the group and cancel more capacity than intended.

This does not let an arbitrary caller modify another user's group because `setConsumed()` still requires the adapter to be authorized for the initiator. The risk is at the adapter API boundary: a frontend, solver, or SDK can produce a valid authorized bundle that leaves phantom standing-offer capacity or burns too much capacity because the route limit dimension and the standing-offer group dimension are conflated.

Recommendation

Bind the group denomination explicitly. For example, add an expected consume dimension to `executeAndConsume()` and require it to match `params.fillIndex`, or split the API into denomination-specific helpers so a unit group, seller-asset group and buyer-asset group cannot be advanced with the wrong total. If the adapter intentionally leaves this responsibility to offchain tooling, document that `consumeGroup` must only be used with a `fillIndex` matching the active max field of every standing offer in that group and add SDK/front-end validation for that invariant.

Resolution

Tenor Team: Acknowledged.

L-02 | Unbound Collateral Supply Denominator

Category	Severity	Location	Status
Validation	Low	MorphoV2SupplyCollateralCallback.sol	Acknowledged

Description

`MorphoV2SupplyCollateralCallback.onSell` decodes `offerSellerAssets` from arbitrary callback data and uses it as the denominator for every pro-rata collateral pull:

```
    CallbackData memory callbackData = abi.decode(data, (CallbackData));
    ...
    uint256 supplyAmount = configAmount.mulDivDown(sellerAssets,
callbackData.offerSellerAssets);
```

The interface documents that `offerSellerAssets` must equal `offer.maxSellerAssets`, but this callback never receives the offer and cannot enforce that relationship. If the offer builder, router or integration signs callback data with a denominator that does not match the offer's actual seller-asset capacity, the callback supplies collateral according to the mismatched denominator instead of the amount implied by the offer. For example, a unit-denominated or price-discounted borrow offer can settle `sellerAssets` below `obligationUnits`. If the callback data is populated from the unit capacity rather than the seller-asset capacity, each fill supplies less collateral than a full-offer simulation using `amounts[]` would suggest.

Midnight's final seller health check still prevents immediately liquidatable debt and a nonzero `maxLtv` adds a stricter post-fill guard. The remaining risk is an integration hazard: users or routers can rely on an unenforced denominator and produce fills with less posted collateral, less liquidation margin or revert-only execution paths than expected.

Recommendation

Only use this callback with seller-asset-denominated offers where `offer.maxSellerAssets` is nonzero and exactly equals `callbackData.offerSellerAssets`. Enforce that invariant in the router, clamp, ratifier or offer-construction layer before a fill is submitted. If the callback is intended to support other offer denominations, include an explicit denomination mode or expected offer hash in the signed callback data and reject mismatches before using the pro-rata amount.

Resolution

Tenor Team: Acknowledged.

L-03 | Renewal Window Can Underflow Before Validation

Category	Severity	Location	Status
Validation	Low	BaseRenewalRatifier.sol	Resolved

Description

`BaseRenewalRatifier._ratifyWindow()` subtracts `p.renewalWindow` from `sourceMaturity` before checking that the renewal window is not larger than the source maturity.

```
    renewalPeriodStart = sourceMaturity - p.renewalWindow;
```

If a user configures `renewalWindow > sourceMaturity`, ratification reverts with Solidity panic `0x11` instead of the ratifier's domain-specific `InvalidRenewalParams()` error. This is reachable because `RenewalIntentRatifier.setParams()` stores user renewal params without validating this relationship.

Recommendation

Validate the configured renewal window before subtracting it from `sourceMaturity`.

```
    if (p.renewalWindow > sourceMaturity) revert InvalidRenewalParams();
    renewalPeriodStart = sourceMaturity - p.renewalWindow;
```

Resolution

Tenor Team: The issue was resolved in PR#489.

L-04 | Optional Router Fee Adjustment

Category	Severity	Location	Status
Validation	Low	TakeRouter.sol	Acknowledged

Description

`TakeRouter._execute` only adjusts `buyerAssets` and `sellerAssets` for callback-level fees when the caller supplies a nonzero `feeAdjuster`. Without that resolver, the router records the raw amounts returned by Midnight and then enforces `maxFill`, `minFill` and price bounds against those raw amounts.

Fee-charging renewal callbacks can still transfer part of those assets to `feeRecipient`. Consequently, a route can satisfy `minFill` on the router's gross accounting while the taker's net amount after the callback fee is lower. The same mismatch can make `maxFill` and aggregate price checks weaker than intended for callbacks with nonzero fees.

This does not bypass the renewal ratifier's rate check and the canonical `CallbackFeeAdjuster` can make the accounting conservative when it is configured correctly. The risk is an integration failure: frontends, keepers or direct callers that omit the matching adjuster receive weaker router-level fill protection than the API shape suggests.

Recommendation

Require a matching fee adjuster for action types and callbacks that can charge nonzero fees or derive the expected fee configuration from callback data and apply the adjustment inside the router. If the router intentionally accepts raw accounting, rename or document `minFill`, `maxFill` and price bounds as raw-Midnight limits rather than net-taker limits.

Resolution

Tenor Team: Acknowledged.

L-05 | Router Limits Assume Homogeneous Actions

Category	Severity	Location	Status
Validation	Low	TakeRouter.sol	Resolved

Description

`TakeRouter._execute` sums `buyerAssets`, `sellerAssets` and `obligationUnits` across every successful action in the batch. It then enforces one `maxFill`, one `minFill` and one average price check against those aggregate totals.

The router does not require the actions to use the same loan token, obligation, maturity, offer side, callback or renewal user. Therefore the batch-level limits are only meaningful when the caller or offchain solver keeps the batch homogeneous. If unrelated actions are mixed, a favorable fill can offset an unfavorable fill and make the aggregate price pass even though one action would not satisfy the same price bound on its own.

This is mainly an integration and user-protection issue. `TAKE_ON_BEHALF` actions still pass through each user's ratifier and `MIDNIGHT_TAKE` acts as the initiating taker. However, if a frontend or solver presents the router's aggregate `maxFill`, `minFill` or price bounds as per-action protection, users can receive weaker protection than expected.

`TakeRouterAdapterBase._resolveSentinel` has the same assumption. It resolves `type(uint256).max` from `actions[0]` only, then applies the resolved cap to the whole batch.

Recommendation

Either enforce homogeneous batches whenever aggregate limits are used or expose per-action bounds so each action can be checked independently.

At minimum, compare every action against the first action's loan token, obligation id and offer side before applying a shared fill cap or shared price bound. If mixed batches are intended, document that router totals are batch-level accounting only and must not be treated as per-action slippage protection.

Resolution

Tenor Team: The issue was resolved in PR#485.

L-06 | Clamp Data Not Bound To Callback Data

Category	Severity	Location	Status
Validation	Low	src/periphery/clamps/SupplyCollateralCallbackClamp.sol#L29	Acknowledged

Description

`SupplyCollateralCallbackClamp` documents (line 29) that `clampData.collateralAmounts` must equal `offer.callbackData.amounts`, but never enforces it on-chain. `TakeRouter._capTakeUnits` passes both blobs independently to the clamp and the callback. If the keeper passes a `clampData` with inflated amounts, `clamp()` returns an inflated `maxUnits`, the take proceeds, and the callback supplies the lower real amount — leaving the seller with debt exceeding the supplied collateral. The only on-chain catch is the callback's optional `maxLtv` check (`MorphoV2SupplyCollateralCallback.onSell`), which defaults to 0 (unconstrained) when `callbackData.length < 160`.

The harmed party is the taker (lender) who ends up holding undercollateralized debt — exposing them to the keeper they authorized.

Recommendation

Document that callers MUST pass `clampData.collateralAmounts` identical to the `amounts` field encoded in `offer.callbackData`, and that with `maxLtv = 0` the only safety net is keeper honesty.

Optionally: have the clamp `abi.decode` `offer.callbackData` and assert array equality before computing bounds. This eliminates the trust assumption entirely.

Resolution

Tenor Team: Acknowledged.

L-07 | Renewal Can Rescue A Liquidatable Obligation

Category	Severity	Location	Status
Trust Assumptions	Low	src/callbacks/MorphoV2ToV2BorrowRepay.sol	Acknowledged

Description

Midnight's `take()` does not check source-obligation health. A renewal whose source is already liquidatable will still execute, atomically repaying source debt and migrating collateral to the target — effectively dodging liquidation. This is inherent to Midnight (the same is achievable via flash loans or hand-rolled callbacks); Tenor's renewal paths just expose it as a packaged flow.

Impact: borrowers (or anyone they authorize) can use renewal to avoid the liquidation incentive penalty. Risk migrates to whichever target obligation the renewal intent points at.

Recommendation

Document explicitly that renewal paths do not block liquidatable sources — they are not "rescue-safe" from the protocol's perspective, and borrowers configuring a more lenient target market can use renewal to bypass liquidation.

Resolution

Tenor Team: Acknowledged.

I-01 | Effective Units Panic On Long Maturities

Category	Severity	Location	Status
Validation	Info	BaseRenewalRatifier.sol	Resolved

Description

`_effectiveUnitsPerWad()` subtracts the target obligation's continuous-fee discount from `WAD` without checking that the product is below `WAD`.

```
    uint256 cf = MORPHO_MIDNIGHT.continuousFee(obligationId);
    if (cf == 0) return WAD;
    uint256 t = UtilsLib.zeroFloorSub(offer.obligation.maturity, block.timestamp);
    return WAD - cf * t;
```

Midnight bounds `continuousFee`, but `t` comes from the target obligation maturity. At the maximum Midnight continuous fee, `cf * t` reaches `WAD` at roughly 100 years. `UserRenewalParams.maxDuration` is a `uint32`, so a user can authorize a target duration above that threshold. The ratifier's target-maturity check only compares the maturity against `[block.timestamp + minDuration, block.timestamp + maxDuration]`. It does not reject maturities where `continuousFee * timeToMaturity > WAD`.

Consequently, a lend renewal into a very long target maturity with a nonzero continuous fee can revert with Solidity arithmetic panic `0x11` during ratification. Midnight would later reject credit creation when pending fees exceed credit, but the ratifier should fail with an intentional validation error instead of an unchecked arithmetic panic.

Recommendation

Reject target maturities whose continuous-fee product is too large before subtracting from `WAD`.

```
    uint256 fee = cf * t;
    if (fee > WAD) revert InvalidTargetMaturity();
    return WAD - fee;
```

Use `>= WAD` instead if zero effective units should also be treated as invalid. This is preferable to silently saturating to zero because the underlying Midnight trade is not useful once continuous fees consume all target credit.

Resolution

Tenor Team: https://github.com/Shippooor-Labs/tenor-morpho-v2-contracts-2/pull/490/.

I-02 | Open Liquidator Choice Delays Liquidations

Category	Severity	Location	Status
Warning	Info	DelayedLiquidationGate.sol	Acknowledged

Description

`startGracePeriod()` can be called by any account once a position is unhealthy and not liquidation-locked. The caller supplies `_priorityLiquidator` and the gate stores that address without checking that it is authorized, borrower-approved, or derived from a deterministic rule.

`_requireLiquidationAllowed()` later enforces the stored address during the first `PRIORITY_PERIOD` after the grace period. While that exclusive window is active, every other liquidator reverts with `LiquidationNotAllowed`.

Consequently, a bot watching unhealthy positions can front-run the first grace-period transaction and assign itself as the priority liquidator. If it does not liquidate, other liquidators must wait until the priority period expires. Factory-created gates currently limit this exclusive window to one minute, so the impact is limited in the intended factory setup. Direct `DelayedLiquidationGate` deployments can choose a longer `PRIORITY_PERIOD`, making the delay materially worse.

Recommendation

Do not let arbitrary callers choose the priority liquidator for a permissionless grace-period start. Either remove priority assignment from `startGracePeriod()`, require an authorized market role to set a nonzero priority liquidator, or derive the priority liquidator from a deterministic market configuration.

Also enforce the intended maximum `PRIORITY_PERIOD` in the `DelayedLiquidationGate` constructor, not only in the factory, so direct deployments cannot bypass the deployment-time cap.

Resolution

Tenor Team: Acknowledged.

I-03 | Rate Limits Ignore Settlement Rounding

Category	Severity	Location	Status
Warning	Info	BaseRenewalRatifier.sol	Acknowledged

Description

`BaseRenewalRatifier._ratifyRate` checks the user's rate limit against a normalized per-WAD price. It passes `effUnitsPerWad` and `effPrice` into `PriceLib.satisfiesRateLimit`, so the check does not know the actual `takeUnits` value or the integer asset amounts that `Midnight.take` will settle.

`Midnight.take` later computes the real `buyerAssets` and `sellerAssets` with mixed integer rounding. For small fills, the rounded settlement price can be materially worse than the normalized price approved by the ratifier. For example, if the theoretical buyer price is `0.5e18` and the fill is `1` unit, `mulDivUp` charges `1` raw asset unit, so the realized price is `1e18`.

This is usually dust for 18-decimal assets. The gap is more relevant for low-decimal loan tokens and repeated tiny fills. In those cases a keeper can execute fills that satisfy the continuous price check while the user receives a worse realized integer price. `UserRenewalParams` also has no minimum fill or minimum asset constraint to let the user reject these rounded executions at the ratifier level.

Recommendation

Add a user-configurable minimum fill size or minimum realized asset amount to the renewal policy, then reject takes below that threshold before calling `Midnight.take`.

Alternatively, add a post-take validation step in `RenewalIntentTakeOnBehalf.take` that checks the returned `buyerAssets`, `sellerAssets` and `obligationUnits` against the user's realized price limits. The validation should use the same rounded values that were actually settled.

Resolution

Tenor Team: Acknowledged.

I-04 | Standing Renewals Lack Amount Bounds

Category	Severity	Location	Status
Warning	Info	IRenewalIntentRatifier.sol	Acknowledged

Description

`UserRenewalParams` records rate, timing, cadence and market constraints. It does not include a maximum asset amount, a maximum unit amount, a one-shot nonce or cumulative consumption for the renewal user. `RenewalIntentRatifier` then reuses those params for every take matching `(user, callback, sourceMarketId, targetMarketId)`. `BaseRenewalRatifier` validates the callback tuple, fee config, renewal window, target maturity and rate policy, but it does not validate the size of the take.

For V1 to V2 lend migrations, the take size directly controls how much `MorphoV1ToV2LendCallback` withdraws from the user's ERC4626 source vault:

```
    IERC4626(callbackData.vault).withdraw(buyerAssets + fee, address(this), buyer);
```

After the user has approved this callback for vault shares, authorized `RenewalIntentTakeOnBehalf`, authorized the ratifier and set params for the V1 to V2 tuple, any keeper can execute as much migration as offer liquidity, vault withdrawability and the callback approval allow. The user still receives target V2 credit at a rate accepted by their policy, so this is not direct theft. It can still surprise users or integrations that treat standing renewal params as consent for a bounded migration amount. A keeper can force the user's whole approved V1 vault position into V2 and charge the configured fees.

Recommendation

Add user-controlled amount bounds to the renewal authorization. At minimum, include a max asset or max unit amount in the params used for each `(user, callback, sourceMarketId, targetMarketId)` tuple and enforce it during ratification. If standing automation is meant to be reusable, track cumulative consumption under `intent.user` and reset it by nonce, epoch or cadence period. If a renewal should be one-shot, include an expiry or nonce and consume it before external callbacks can run.

Resolution

Tenor Team: Acknowledged.

I-05 | Vault Withdraw Callback Allows Position Crossing

Category	Severity	Location	Status
Warning	Info	MorphoV2WithdrawVaultSharesCallback.sol	Acknowledged

Description

`MorphoV2WithdrawVaultSharesCallback.onBuy()` does not check whether the BUY fill crosses the buyer from debt into credit. The callback withdraws vault-share collateral, redeems the shares for `buyerAssets`, approves Midnight and returns success without checking the buyer's post-fill position.

Midnight allows this crossing when the signed offer is not `reduceOnly`. That means a stale or oversized vault-withdraw BUY offer can repay the borrower's remaining debt, redeem collateral and leave the borrower with a new credit position on the same obligation. `MorphoV2ToV1BorrowCallback.onBuy()` rejects the same condition with `PositionCrossing`, but this callback does not.

This may be economically valid for some users, but it contradicts the borrower early-exit intent of this callback. It also makes stale offers more dangerous: if the borrower's live debt falls below the signed fill size before execution, the callback can still complete and convert the excess into credit instead of reverting.

Recommendation

Add the same post-fill crossing guard used by `MorphoV2ToV1BorrowCallback`. After approving Midnight, call `updatePositionView()` for the buyer and revert with `CallbackLib.PositionCrossing()` if the resulting credit is nonzero.

Alternatively, require all integrations that use this callback to cap take units to the buyer's current debt before calling Midnight. The safer default is to enforce the invariant inside the callback so direct Midnight fills and unclamped router calls receive the same protection.

Resolution

Tenor Team: Acknowledged.

I-06 | Stale MAX_FEE_RATE_V2_V1 In Docs

Category	Severity	Location	Status
Documentation	Info	src/libraries/Constants.sol#L8	Resolved

Description

`DIMENSIONAL_UNITS.md` documents `MAX_FEE_RATE_V2_V1 = 0.0025e18 (0.25% with units D18{loan-token/borrow-token/sec}). The actual constant in src/libraries/Constants.sol` is `0`, meaning no fee component is permitted on the V2→V1 renewal direction.

Integrators reading the doc to size off-chain rate calculations will use a fee assumption the contract rejects, producing renewal attempts that always revert at `BaseRenewalRatifier._ratifyRate`.

Recommendation

Update `DIMENSIONAL_UNITS.md` so `MAX_FEE_RATE_V2_V1 = 0` matches the on-chain constant. Optionally add a one-line note explaining why V2→V1 has a hard-zero fee while V2→V2 and V1→V2 have non-zero ceilings.

Resolution

Tenor Team: The issue was resolved in PR#456.

I-07 | allowRevert Scope Is Incomplete

Category	Severity	Location	Status
Documentation	Info	src/periphery/TakeRouter.sol	Acknowledged

Description

`Action.allowRevert = true` only catches reverts inside the inner `_MIDNIGHT.take` / `_TAKE_ON_BEHALF.take` `try/catch`. The following per-action paths run outside the `try/catch` and abort the entire batch despite `allowRevert`:

1. `abi.decode(action.data, ...)` — malformed action data (L:264, 297).
2. `_MIDNIGHT.touchObligation(action.offer.obligation)` — invalid obligation (L:266, 299).
3. `ICallbackFeeAdjuster.beforeDispatch(...)` — arbitrary external call (L:347).
4. `RouterLib.budgetToUnits(...)` — arithmetic (L:349).
5. `ClampLib.getOfferRemaining(...)` — arithmetic (L:355).
6. `ITakeClamp.clamp(...)` — arbitrary external call (L:358).
7. `ICallbackFeeAdjuster.afterDispatch(...)` — arbitrary external call, runs after a successful dispatch (L:206).

Integrators relying on `allowRevert = true` for batch resiliency may not realize that these paths still abort the whole batch.

Recommendation

Update the `Action.allowRevert` natspec and the file-level comment block to enumerate the pre-dispatch and post-dispatch revert paths that bypass it. For the external calls and decode paths, consider wrapping them in their own `try/catch` so `allowRevert` actually covers them.

Resolution

Tenor Team: Acknowledged.

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian

Founded in 2022 by DeFi experts, Guardian is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit https://guardianaudits.com

To view our audit portfolio, visit https://github.com/guardianaudits

To book an audit, message https://t.me/guardianaudits