Blackthorn

Security Review For **Tenor Labs**

Collaborative Audit Prepared For:	**Tenor Labs**
Lead Security Expert(s):	**0x73696d616f**
	0xapple
	hyh
	xiaoming90
Date Audited:	**May 22 - June 5, 2026**

Introduction

Tenor Contracts is a collection of immutable contracts that extend the Morpho stack, primarily Morpho Midnight, Morpho's intent-based fixed-rate lending protocol, alongside Morpho Blue, Morpho Vault V2, and Morpho Oracles.

The contracts add stateless callbacks invoked during Morpho Midnight's take(), each encoding an atomic state transition: migrations to and from Morpho Blue and Vault V2, and Midnight-to-Midnight renewals. Together with these migration callbacks, a Migration Ratifier lets users set policies for how their positions can be renewed or migrated on their behalf, and validates each counterparty's offer against the configured parameters. A batch-fill router, exposed through a Bundler3 adapter, can be used to fill a position across multiple offers in one transaction.

The repository also includes standalone contracts, including a delayed-liquidation gate and an oracle that implements Morpho's oracle interface and cross-checks its price against a secondary feed for added robustness.

Scope

Repository: Shippooor-Labs/tenor-morpho-v2-contracts-2

Audited Commit: dbe3a089eb0728a0ec77c23ecf393fff028cb468

Final Commit: 7ce125563f32c34069dd4ab1a6c3f6764ac21529

Files:

- src/BaseMigrationRatifier.sol
- src/bundler/AuthorizationAdapter.sol
- src/bundler/MidnightAdapterBase.sol
- src/bundler/MigrationIntentRatifierAdapterBase.sol
- src/bundler/TenorAdapter.sol
- src/bundler/TenorRouterAdapterBase.sol
- src/callbacks/BorrowBlueToMidnightCallback.sol

- src/callbacks/BorrowMidnightRenewalCallback.sol
- src/callbacks/BorrowMidnightToBlueCallback.sol
- src/callbacks/LendMidnightRenewalCallback.sol
- src/callbacks/LendMidnightToVaultCallback.sol
- src/callbacks/LendVaultToMidnightCallback.sol
- src/callbacks/MidnightSupplyCollateralCallback.sol
- src/callbacks/MidnightSupplyVaultSharesCallback.sol
- src/callbacks/MidnightWithdrawVaultSharesCallback.sol
- src/factories/DelayedLiquidationGateFactory.sol
- src/factories/MidnightAllowlistGateFactory.sol
- src/factories/OracleWithValidationFactory.sol
- src/factories/VaultV2AllowlistGateFactory.sol
- src/gates/DelayedLiquidationGate.sol
- src/gates/MidnightAllowlistGate.sol
- src/gates/VaultV2AllowlistGate.sol
- src/IntentSettler.sol
- src/interfaces/IIntentRatifier.sol
- src/interfaces/IIntentSettler.sol
- src/interfaces/IMigrationIntentRatifier.sol
- src/libraries/CallbackLib.sol
- src/libraries/CollateralTransferLib.sol
- src/libraries/Constants.sol
- src/libraries/LinearInterpolationLib.sol
- src/libraries/PriceLib.sol

- src/libraries/RatePointLib.sol
- src/libraries/TenorMarketIdLib.sol
- src/MigrationIntentRatifier.sol
- src/oracles/OracleWithValidation.sol
- src/periphery/interfaces/ICallbackFeeAdjuster.sol
- src/periphery/interfaces/IMidnightVaultExecutor.sol
- src/periphery/interfaces/ITakeClamp.sol
- src/periphery/interfaces/ITenorRouter.sol
- src/periphery/libraries/ClampLib.sol
- src/periphery/libraries/MidnightLib.sol
- src/periphery/libraries/RouterLib.sol
- src/periphery/MidnightVaultExecutor.sol
- src/periphery/resolvers/CallbackFeeAdjuster.sol
- src/periphery/TenorRouter.sol
- src/policies/FourWeekCadence.sol
- src/policies/MarketMakingPolicy.sol
- src/policies/PausableBase.sol
- src/policies/PausableMarketMakingPolicy.sol
- src/policies/PausableStaticRatePolicy.sol
- src/policies/StaticRatePolicy.sol

Final Commit Hash

7ce125563f32c34069dd4ab1a6c3f6764ac21529

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	2	25

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Security Experts Dedicated to This Review

@0x73696d616f

@0xapple

@hyh

@xiaoming90

Issue H-1: `DelayedLiquidationGate.onLiquidate()` ignores the Midnight-supplied liquidator [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/68

Vulnerability Detail

Liquidator performing liquidation via the `DelayedLiquidationGate` will need to grant allowance to the `DelayedLiquidationGate` address because the gate will pull the repayment from the liquidator (caller) in the `DelayedLiquidationGate.onLiquidate` function.

Assume that there are some liquidators that grant the maximum allowance to the `DelayedLiquidationGate` address because they do not want to re-grant the allowance every single time they want to perform a liquidation. As such, there might be some residual allowance left.

However, the issue is that the `DelayedLiquidationGate.onLiquidate()` is not bound to calls initiated by `DelayedLiquidationGate.liquidate()`. Midnight passes the original Midnight liquidator as the `liquidator` callback argument, but the gate ignores this parameter, and does not perform any validation against it.

```
File: DelayedLiquidationGate.sol
096:     function onLiquidate(
097:         bytes32 marketId,
098:         Market memory market,
099:         uint256 collateralIndex,
100:         uint256 seizedAssets,
101:         uint256 repaidUnits,
102:         uint256 badDebt,
103:         address, /* liquidatorFromMidnight */
104:         address borrower,
105:         address receiver,
```

```
106:         bytes memory data
107:     ) external returns (bytes32) {
```

As a result, any user can call `MIDNIGHT.liquidate()` on any market where they are allowed to liquidate, set:

```
receiver = address(delayedLiquidationGate)
callback = address(delayedLiquidationGate)
data = abi.encode(victim, "")
```

Then Midnight will call the gate's `onLiquidate()`, and the gate will decode `victim` as the liquidator and execute:

```
SafeTransferLib.safeTransferFrom(market.loanToken, liquidator, address(this),
↪  repaidUnits);
IERC20(market.loanToken).forceApprove(address(MIDNIGHT), repaidUnits);
```

This lets an attacker spend loan-token allowance that a victim has granted to the delayed liquidation gate, even though the victim did not call `DelayedLiquidationGate.liquidate()` and did not authorize this specific liquidation.

Assume the following scenario:

1. Victim has approved the delayed liquidation gate for a loan token (e.g., USDC)
2. Attacker creates or uses an ungated Midnight market with that same loan token, attacker-controlled collateral, and attacker-controlled borrower/lender positions.
3. Attacker calls `MIDNIGHT.liquidate()` directly with the delayed liquidation gate as `receiver` and `callback`, encoding the victim as the liquidator.
4. Gate transfers seized collateral to the victim, then pulls `repaidUnits` USDC from the victim using the victim's allowance.
5. The liquidation is completed, and the victim ends up receiving worthless or attacker-controlled collateral while losing real loan tokens.

Impact

Any attacker can call `MIDNIGHT.liquidate()` directly with the gate as `receiver/callback` and a victim encoded as the liquidator, spending the victim's loan-token allowance to the gate without authorization. The victim loses real loan tokens and receives worthless or attacker-controlled collateral.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/b6b2f31b18e09f2b411ff8589f79c26e20d67b33/DelayedLiquidationGate.sol#L96

Tool Used

Manual Review

Recommendation

Consider adding `if (liquidatorFromMidnight != address(this)) revert InvalidLiquidationCallback();` check to the gate's `onLiquidate()`.

Issue M-1: Rate guard checks the wrong market and curve side on the maker-side path [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/65

Vulnerability Detail

The `MarketMakingPolicy` stores one curve per Midnight market per user.

To validate an offer, it must select two things: which side's market/maturity to look up, and which rate (buy vs sell). It derives both from the raw offer.buy flag forwarded by `BaseMigrationRatifier._ratifyRate()`:

```
(bytes32 marketId, uint256 maturity) =
    offerIsBuy ? (sourceTenorMarketId, sourceMaturity) : (targetTenorMarketId,
    ↪  targetMaturity);
..SNIP..
(uint256[] memory knots, uint256[] memory values) = _loadCurve(curve, offerIsBuy);
```

So `offerIsBuy == true` maps to `(source, sellRate)`, and `offerIsBuy == false` maps to `(target, buyRate)`.

Source and target come from the migration direction: source is the venue the position moves out of, target is the venue it moves into.

- Entering Midnight (`LEND_VAULT_TO_MIDNIGHT`, `BORROW_BLUE_TO_MIDNIGHT`): the Midnight market is the target.
- Exiting Midnight (`LEND_MIDNIGHT_TO_VAULT`, `BORROW_MIDNIGHT_TO_BLUE`): the Midnight market is the source.

It works on `IntentSettler.take()` (user is taker) because the user fills a counterparty's offer, so `offer.buy` is the counterparty's direction, which is the opposite of the user's.

- Lend entry (vault to Midnight): user lends/buys, so they fill a sell offer, `offer.buy == false`, table picks target = the Midnight market, and buyRate, which is correct.
- Lend exit (Midnight to vault): user exits/sells, so they fill a buy offer, `offer.buy == true`, table picks source = the Midnight market, and sellRate, which is correct.

However, this breaks on `IntentSettler.isRatified()` (user is maker). Here, the user posted the offer, so `offer.buy` equals the user's own direction.

The migration geometry is unchanged (Midnight is still target on entry, source on exit), but the mapping now points at the wrong side:

- Lend entry (vault to Midnight) as maker: user lends/buys, so they post a buy offer, `offer.buy == true`, table picks source = the vault and sellRate, instead of the Midnight target and buyRate.
- Lend exit (Midnight to vault) as maker: user exits/sells, so they post a sell offer, `offer.buy == false`, table picks target = the vault and buyRate, instead of the Midnight source and sellRate.

So, on the maker-side path, the rate guard checks the wrong market and curve side, so it either reverts valid intents (`NoCurveForUserMarket`) or approves offers at rates worse than the maker's configured curve.

Impact

Valid intents will revert or offers at rates worse than the maker's configured curve will be approved.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/27c7b68bf46f067689f552771c3f2773663fdb11/MarketMakingPolicy.sol#L62

Tool Used

Manual Review

Recommendation

Consider updating the `IInterestRatePolicy.getRate()` to ensure that the rate guard checks the correct market and curve side for both taker and maker paths.

Issue M-2: Maker-side intents are ratified with taker-side trading fees leading to valid orders being incorrectly rejected [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/72

Vulnerability Detail

In the `IntentSettler`, Tenor supports two settlement paths:

- `take()` is the user-as-taker path. It calls the user's ratifier before forwarding to `MORPHO_MIDNIGHT.take(...)` with `taker = intent.user`.
- `isRatified()` is the user-as-maker path. Midnight calls this when the user is the offer maker, and `IntentSettler` checks `intent.user == offer.maker`.

Therefore, a user's intent can be settled either as a Midnight taker or as a Midnight maker.

The `_ratifyRate` picks the effective price purely from `_userIsBuy(callback)`:

```
uint256 effPrice = userIsBuy
    ? RouterLib.netBuyerPrice(tickPrice, tradingFee, feeRateForRateCheck)
    : RouterLib.netSellerPrice(tickPrice, tradingFee, feeRateForRateCheck);
```

`RouterLib.netBuyerPrice()` assumes the user is a buyer-as-taker:

```
uint256 midnightPrice = offerPrice + tradingFee;
if (feeRate == 0) return midnightPrice;
uint256 tenorPrice = CallbackLib.buyerEffectivePrice(offerPrice, feeRate);
return midnightPrice > tenorPrice ? midnightPrice : tenorPrice;
```

`RouterLib.netSellerPrice()` assumes the user is a seller-as-taker:

```
uint256 midnightPrice = offerPrice > tradingFee ? offerPrice - tradingFee : 0;
if (feeRate == 0) return midnightPrice;
```

```
uint256 tenorPrice = CallbackLib.sellerEffectivePrice(offerPrice, feeRate);
return midnightPrice < tenorPrice ? midnightPrice : tenorPrice;
```

This is correct for the taker path, but it is not always correct for the maker path.

The Midnight charges `tradingFee` to the taker, not maker. Below is from the Midnight contract.

```
uint256 sellerPrice = offer.buy ? offerPrice - _tradingFee : offerPrice;
uint256 buyerPrice = sellerPrice + _tradingFee;
```

The following shows the effective price used by the code.

user side	`offer.buy`	role	actual Midnight price	code uses
buyer	true	maker buyer	`tickPrice`	`tickPrice + fee` (Wrong)
buyer	false	taker buyer	`tickPrice + fee`	`tickPrice + fee` (Correct)
seller	false	maker seller	`tickPrice`	`tickPrice - fee` (Wrong)
seller	true	taker seller	`tickPrice - fee`	`tickPrice - fee` (Correct)

As a result, when the user is the maker, `_ratifyRate()` applies the Midnight trading fee as if the user were the taker.

The full effective-price comparison including Tenor callback fees is as follows:

user side	maker actual effective price	ratifier effective price
maker buyer	`max(tickPrice, tenorBuyerPrice)`	`max(tickPrice + tradingFee, tenorBuyerPrice`
maker seller	`min(tickPrice, tenorSellerPrice)`	`min(tickPrice - tradingFee, tenorSellerPric`

Therefore, the ratifier can be stricter than execution for maker-side intents, and a valid order might be incorrectly rejected and reverted.

Impact

Valid orders might be incorrectly rejected and reverted.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/9daeed41907f262d18c099af80599e8c323271df/BaseMigrationRatifier.sol#L299

Tool Used

Manual Review

Recommendation

Consider adjusting the `_ratifyRate()` function to price maker-side intents without Midnight taker trading fees, using `offer.buy`/caller role to distinguish maker vs taker execution.

Issue L-1: MidnightSupplyVaultSharesCallback deposits into ERC4626 without a min shares slippage bound [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/52

Summary

`MidnightSupplyVaultSharesCallback::onSell()` deposits the borrower's loan tokens into the configured ERC4626 vault and supplies the resulting shares as collateral, without enforcing any minimum shares received. The borrower has no protection against vault share price drift between offer creation and offer fill. Same for other vault callbacks.

Vulnerability Detail

The callback's deposit accepts whatever shares the vault returns:

```
IERC20(loanToken).forceApprove(vault, totalDeposit);
uint256 shares = IERC4626(vault).deposit(totalDeposit, address(this));

IERC20(vault).forceApprove(address(MORPHO_MIDNIGHT), shares);
MORPHO_MIDNIGHT.supplyCollateral(market, callbackData.collateralIndex, shares,
↪  seller);
```

The borrower configures `CallbackData` when creating the sell offer. The offer sits open until a taker fills it, which may happen much later. When `additionalDepositPercent > 0`, fresh borrower funds are also pulled in via `safeTransferFrom`, so the borrower's own capital flows into the vault at the share price observed at fill time.

If the vault's share price degrades between offer creation and offer fill (vault loss, oracle drift, hostile curator action), the borrower receives fewer shares than expected, supplying less collateral than intended and ending up with a smaller LTV buffer than they signed up for.

The user-facing `MidnightVaultExecutor` enforces `maxSharePriceE27` for the exact same flow (reference), but the offer-fill callback omits it.

Impact

The borrower's collateral position is exposed to ERC4626 share price drift with no cap, weakening their health factor relative to what the offer parameters implied.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/main/tenor-morpho-v2-contracts-2/src/callbacks/MidnightSupplyVaultSharesCallback.sol#L65

Tool Used

Manual Review

Recommendation

Add a `maxSharePriceE27` (or `minShares`) field to `CallbackData` and revert when the realized share price exceeds it, matching the `MidnightVaultExecutor` pattern. Or document that the vault is safe.

Issue L-2: MidnightVaultExecutor::onLiquidate forces liquidators to provide callback data and never refunds leftover loan tokens [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/53

Summary

`MidnightVaultExecutor::onLiquidate()` always sends the redeemed loan tokens to the `liquidator` and invokes their callback only when `callbackData.length > 0`. A liquidator without a callback contract has no way to push the funds back to the executor for Midnight to pull the debt, so the liquidation reverts. The sibling `onRepay` already supports the no-callback path. Leftover loan tokens are also never swept to the liquidator on this path.

Vulnerability Detail

In onLiquidate, the redeemed underlying is unconditionally sent to the liquidator:

```
uint256 redeemed = IERC4626(vault).redeem(seizedShares, liquidator, address(this));

if (minSharePriceE27 != 0 && UtilsLib.mulDivDown(redeemed, RAY, seizedShares) <
↪  minSharePriceE27) {
    revert SlippageExceeded();
}

if (callbackData.length > 0) {
    require(
        ILiquidateCallback(liquidator).onLiquidate(...) == CALLBACK_SUCCESS,
        IMidnight.WrongLiquidateCallbackReturnValue()
    );
```

```
}
```

After `onLiquidate` returns, Midnight pulls `repaidUnits` of loan token from the executor. With `callbackData.length == 0`, no one has transferred the loan token back to the executor, so the pull reverts and the liquidation fails. The only liquidators that work are ones that implement `ILiquidateCallback`.

Compare with onRepay, which routes the redeem recipient based on whether callback data is present:

```
address recipient = callbackData.length > 0 ? caller : address(this);
uint256 redeemed = IERC4626(vault).redeem(sharesToWithdraw, recipient,
↪  address(this));
```

Additionally, repayAndWithdrawCollateral sweeps leftover loan tokens back to the caller, but `liquidateAndRedeem` and `onLiquidate` do not. Any loan-token dust left on the executor after the liquidation is forfeited and may be swept by the next caller.

Impact

EOA liquidators and any liquidator without a Midnight-compatible callback contract are DoSed from liquidating vault-share collateral through the executor. Any leftover loan tokens on the executor after a successful liquidation are forfeited.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/main/tenor-morpho-v2-contracts-2/src/periphery/MidnightVaultExecutor.sol#L222

Tool Used

Manual Review

Recommendation

Mirror the `onRepay` pattern: redeem to `address(this)` when there is no callback data, and only forward to the liquidator inside the callback branch. After the liquidation completes

in `liquidateAndRedeem`, transfer the redeemed loan token and any dust to the liquidator:

```
address recipient = callbackData.length > 0 ? liquidator : address(this);
uint256 redeemed = IERC4626(vault).redeem(seizedShares, recipient, address(this));
```

And in `liquidateAndRedeem`, sweep the remaining loan-token balance to `msg.sender` after the `MORPHO_MIDNIGHT.liquidate` call returns.

Issue L-3: ClampLib::mulDivDownInverse reverts on `target == type(uint256).max` before the overflow guard runs [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/54

Summary

`mulDivDownInverse` computes `n = target + 1` before its overflow guard, so `target == type(uint256).max` reverts with arithmetic overflow instead of returning `type(uint256).max` as the guard's intent implies.

Vulnerability Detail

The function:

```
function mulDivDownInverse(uint256 target, uint256 den, uint256 num) internal pure
↪  returns (uint256) {
    uint256 n = target + 1;
    if (n == 0 || den > type(uint256).max / n) {
        return type(uint256).max; // overflow -> this constraint is not the
        ↪  bottleneck
    }
    return (n * den - 1) / num;
}
```

With Solidity 0.8.x checked arithmetic, `target + 1` reverts when `target == type(uint256).max`. The guard immediately below is meant to handle the overflow case by returning `type(uint256).max`, but it never executes.

`getOfferRemaining` reaches this function with `remainingAssets` derived from `offer.maxAssets`, which is a `uint256` controlled by the maker. A maker setting `offer.maxAssets = type(uint256).max` (or any value such that `remainingAssets` equals `type(uint256).max`) DoSes any helper that calls `getOfferRemaining` for that offer, including the router's clamp logic.

Impact

Callers reading offer state for a maker-controlled `offer.maxAssets == type(uint256).max` revert instead of receiving the documented "unbounded" sentinel, blocking offer reads and any clamp-based path that consumes them.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/main/tenor-morpho-v2-contracts-2/src/periphery/libraries/ClampLib.sol#L233

Tool Used

Manual Review

Recommendation

Short-circuit before the addition:

```
function mulDivDownInverse(uint256 target, uint256 den, uint256 num) internal pure
↪  returns (uint256) {
    if (target == type(uint256).max) return type(uint256).max;
    uint256 n = target + 1;
    if (den > type(uint256).max / n) {
        return type(uint256).max;
    }
    return (n * den - 1) / num;
}
```

Issue L-4: ClampLib::mulDivDownInverse contains a dead `n == 0` check [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/55

Summary

The `n == 0` branch in `mulDivDownInverse` is unreachable: `n` is assigned `target + 1`, so it is always `>= 1` in any path that reaches the check (the only way `n` could be `0` is the overflow case `target == type(uint256).max`, which already reverts on the line above under Solidity 0.8.x checked arithmetic).

Vulnerability Detail

Code:

```
uint256 n = target + 1;
if (n == 0 || den > type(uint256).max / n) {
    return type(uint256).max;
}
```

`n == 0` looks like an intentional overflow guard, but with checked arithmetic the addition reverts before `n` becomes `0`. The check is misleading and obscures the real overflow case (see the sibling finding on `target + 1` overflowing).

Impact

Confusing code.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/main/tenor-morpho-v2-contracts-2/src/periphery/libraries/ClampLib.sol#L234

Tool Used

Manual Review

Recommendation

```
if (target == type(uint256).max) return type(uint256).max;
uint256 n = target + 1;
if (den > type(uint256).max / n) return type(uint256).max;
```

Issue L-5: ClampLib::mulDivUpInverse overflows on large `target`, DoSing SELL offer reads [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/56

Summary

`mulDivUpInverse` computes `target * den` via `mulDivDown` without an overflow guard. Since `den == WAD` and `num == price` (typically `num <= den`), `target * WAD` overflows for `target > type(uint256).max / WAD`. The sibling `mulDivDownInverse` does guard this case and returns `type(uint256).max`.

Vulnerability Detail

The helper:

```
function mulDivUpInverse(uint256 target, uint256 den, uint256 num) internal pure
↪  returns (uint256) {
    return target.mulDivDown(den, num);
}
```

`getOfferRemaining` calls this with `target = remainingAssets` (derived from `offer.maxAssets`) and `den = WAD = 1e18` for SELL offers:

```
uint256 units =
    offer.buy ? mulDivDownInverse(remainingAssets, WAD, price) :
    ↪  mulDivUpInverse(remainingAssets, WAD, price);
```

`offer.maxAssets` is a `uint256` chosen by the maker. Any value above `~1.16e59` (`type(uint256).max / 1e18`) overflows inside `mulDivDown`, reverting the entire `getOfferRemaining` call. `mulDivDownInverse` handles the symmetric situation gracefully by returning `type(uint256).max` and capping further via `UtilsLib.min(units, type(uint128).max)`, but `mulDivUpInv`

`erse` does not.

A SELL offer maker who sets `offer.maxAssets` above this threshold (or `type(uint256).max` as an "unbounded" sentinel) DoSes any path that reads remaining capacity for the offer.

Impact

SELL offers with sufficiently large `maxAssets` revert on any clamp read, breaking the router's clamp path and any external integration that calls `getOfferRemaining` for the offer.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/main/tenor-morpho-v2-contracts-2/src/periphery/libraries/ClampLib.sol#L245

Tool Used

Manual Review

Recommendation

Mirror the guard from `mulDivDownInverse`:

```
function mulDivUpInverse(uint256 target, uint256 den, uint256 num) internal pure
↪  returns (uint256) {
    if (target == 0) return 0;
    if (den > type(uint256).max / target) return type(uint256).max;
    return target.mulDivDown(den, num);
}
```

Issue L-6: Zero Return From Validation Oracle Bypasses Fallback Logic, Causing Unintended Reverts in `OracleWithValidation` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/58

Summary

`OracleWithValidation` only treats validation oracle failures as reverts. If the validation oracle instead signals failure by returning `0`, the contract performs the deviation check against an invalid price and reverts with `ExcessiveOracleDeviation` rather than falling back to the primary oracle. This can cause health checks and liquidations to fail despite `REVERT_ON_VALIDATION_ORACLE_FAILURE` being disabled.

Vulnerability Detail

`OracleWithValidation.price()` is intended to gracefully fall back to `primaryPrice` when the validation oracle is unavailable and `REVERT_ON_VALIDATION_ORACLE_FAILURE` is set to `false`.

However, this behavior is only triggered when `VALIDATION_ORACLE.price()` reverts and execution enters the `catch` block. The implementation therefore assumes that the validation oracle signals failure only by reverting:

```
try VALIDATION_ORACLE.price() returns (uint256 validationPrice) {
    ...
} catch {
    if (REVERT_ON_VALIDATION_ORACLE_FAILURE) revert ValidationOracleFailure();
}
```

This assumption is not enforced by the `IOracle` interface. An oracle implementation may

instead return `0` when it is unable to provide a price rather than reverting. Returning `0` on oracle failure is an established pattern; for example, Compound-style price oracles explicitly return `0` when an invalid price is received from the underlying feed.

In such a scenario, the call succeeds and execution enters the `try` branch. The returned value of `0` is then used in the deviation check:

```
uint256 absoluteDeviation =
    primaryPrice > validationPrice
        ? primaryPrice - validationPrice
        : validationPrice - primaryPrice;
```

Notably, the contract explicitly rejects a primary price of `0`:

```
if (primaryPrice == 0) revert ZeroPrimaryPrice();
```

indicating that `0` is not considered a valid price. However, no equivalent validation is performed on the validation oracle price.

Consider the case where `REVERT_ON_VALIDATION_ORACLE_FAILURE` is set to `false`, `primaryPrice` is 100, and the validation oracle is unavailable and returns `0`. Since the validation oracle call succeeds, execution never reaches the `catch` block. Instead, the deviation is calculated between 100 and 0, causing the transaction to revert with `ExcessiveOracleDeviation`.

As a result, the oracle can revert even though `REVERT_ON_VALIDATION_ORACLE_FAILURE` is disabled, preventing the intended fallback to `primaryPrice`.

Impact

When the validation oracle returns `0` on failure, every `price()` call reverts with `ExcessiveOracleDeviation` for the entire duration of the outage – freezing borrowing, withdrawals, and liquidations regardless of whether the primary oracle is functioning correctly. Positions that become undercollateralized during the freeze cannot be liquidated, accumulating bad debt that may outlast the oracle recovery.

More critically, this silently inverts the intent of `REVERT_ON_VALIDATION_ORACLE_FAILURE = false`. The operator has explicitly configured the protocol to survive validation oracle failures; this bug ensures it does not.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/oracles/OracleWithValidation.sol#L69-L77

Tool Used

Manual Review

Recommendation

Treat a validation oracle price of `0` as a validation oracle failure and route it through the same logic as the `catch` path. This ensures that the behavior of `REVERT_ON_VALIDATION_ORACLE_FAILURE` remains consistent regardless of whether the validation oracle signals failure by reverting or by returning `0`.

Issue L-7: Price Reconstruction From Rounded Assets Causes Valid Orders to Revert in `Router._execute()` [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/59

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

Summary

`Router._execute()` reconstructs execution prices from asset amounts that were already rounded during `Midnight.take()`. Because this round-trip conversion is **non-reversible**, the reconstructed `priceCeil` and `priceFloor` can exceed the actual execution price used during matching.

Vulnerability Details

`Router._execute()` validates slippage by reconstructing an execution price from the asset amount returned by `Midnight.take()`:

```
uint256 priceCeil  = mulDivUp(assets, WAD, units);
uint256 priceFloor = mulDivDown(assets, WAD, units);

if (priceCeil > params.maxPrice) {
    revert PriceSlippageExceeded(priceCeil, params.minPrice, params.maxPrice);
}

if (priceFloor < params.minPrice) {
    revert PriceSlippageExceeded(priceFloor, params.minPrice, params.maxPrice);
}
```

However, the asset amount used for this validation was itself produced using integer division inside `Midnight.take()`:

```
// offerBuy == true
buyerAssets = units.mulDivDown(buyerPrice, WAD);
```

The transformation is not reversible. During asset calculation, `mulDivDown()` truncates the remainder:

```
assets = floor(units * buyerPrice / WAD)
```

When `_execute()` later reconstructs the price using:

```
priceCeil = mulDivUp(assets, WAD, units);
```

it computes:

```
priceCeil = ceil(floor(units * buyerPrice / WAD) * WAD / units)
```

which is not guaranteed to equal `buyerPrice`.

Whenever the original multiplication leaves a non-zero remainder, the truncation performed in `take()` can be partially undone by the upward rounding in `_execute()`, causing the reconstructed price to exceed the price that was originally used to execute the trade.

As a result, slippage validation is performed against a value derived from a rounded intermediate result rather than against the actual execution price agreed upon during matching.

Impact

A buyer who sets `maxPrice = buyerPrice`–the tightest bound that should always be valid–can have their transaction reverted by `_execute()` even though `take()` executed at exactly `buyerPrice`. The seller-side path has the same issue, allowing `minPrice = sellerPrice` orders to revert due solely to rounding introduced during price reconstruction.

This causes valid orders to become unfillable when users specify tight slippage bounds, resulting in unnecessary transaction failures.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/periphery/TenorRouter.sol#L259-L270

Tool Used

Manual Review

Recommendation

Do not reconstruct prices from rounded asset amounts. Instead, propagate the execution prices used during `take()` through `_dispatch()` into `_execute()` and perform slippage validation directly against those values. This removes the lossy round-trip conversion and ensures validation is performed against the actual execution prices.

POC

Run the test with: `forge test --match-test testFuzz_wrongfulTerminationFromPriceCeil -vvv`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";

contract PriceRoundingFuzzTest is Test {
    uint256 constant WAD = 1e18;

    error FillOvershoot(uint256 got, uint256 max);
    error PriceSlippageExceeded(uint256 got, uint256 min, uint256 max);

    struct Result {
        uint256 buyerAssets;
        uint256 sellerAssets;
        uint256 units;
        uint256 priceFloor;
```

```
    uint256 priceCeil;
}

// ---------------------------------------------------------------------------
// Mirrors UtilsLib.mulDivDown / mulDivUp. Inlined for self-containment.
// ---------------------------------------------------------------------------
function mulDivDown(uint256 x, uint256 y, uint256 d) internal pure returns
↪  (uint256) {
    return (x * y) / d;
}

function mulDivUp(uint256 x, uint256 y, uint256 d) internal pure returns
↪  (uint256) {
    return (x * y + d - 1) / d;
}

// ---------------------------------------------------------------------------
// Mirrors the buyerAssets / sellerAssets calculation in Midnight.take().
//
//   buyerAssets  = offerBuy ? mulDivDown(units, buyerPrice,  WAD)
//                           : mulDivUp(units,   buyerPrice,  WAD)
//   sellerAssets = offerBuy ? mulDivDown(units, sellerPrice, WAD)
//                           : mulDivUp(units,   sellerPrice, WAD)
// ---------------------------------------------------------------------------
function _computeTake(
    bool offerBuy,
    uint256 units,
    uint256 sellerPrice,
    uint256 buyerPrice
) internal pure returns (uint256 buyerAssets, uint256 sellerAssets) {
    buyerAssets = offerBuy
        ? mulDivDown(units, buyerPrice, WAD)
        : mulDivUp(units, buyerPrice, WAD);

    sellerAssets = offerBuy
        ? mulDivDown(units, sellerPrice, WAD)
        : mulDivUp(units, sellerPrice, WAD);
}
```

```
// ---------------------------------------------------------------------------
// Checks that the price ceil/floor reconstruction in Router._execute()
// does not exceed/undershoot the price used in Midnight.take().
//
// Mirrors Router._execute() slippage validation:
//   priceCeil  = mulDivUp(assets, WAD, units);
//   priceFloor = mulDivDown(assets, WAD, units);
//   if (priceCeil > params.maxPrice) revert PriceSlippageExceeded(...)
//   if (priceFloor < params.minPrice) revert PriceSlippageExceeded(...)
//
// Price range constrained to [0.7, 1.0) WAD - realistic for Midnight
// lending markets. Below 0.7 WAD, assets can collapse to 0 or 1 for
// small unit counts (separate dust precision issue, not under test here).
// ---------------------------------------------------------------------------
function testFuzz_wrongfulTerminationFromPriceCeil(
    bool offerBuy,
    uint96 units,
    uint96 sellerPrice,
    uint96 fee
) public {
    vm.assume(units > 1e6);
    vm.assume(sellerPrice > 7 * WAD / 10);
    vm.assume(sellerPrice < WAD);
    vm.assume(fee < 1e12); // small relative to price to isolate rounding gap

    uint256 buyerPrice = uint256(sellerPrice) + uint256(fee);

    (uint256 buyerAssets, uint256 sellerAssets) = _computeTake(
        offerBuy,
        uint256(units),
        uint256(sellerPrice),
        buyerPrice
    );

    // Matches sideAssetsIndex selection in Router._execute()
    uint256 assets = offerBuy ? buyerAssets : sellerAssets;
```

```
        if (assets == 0) return; // dust case, skip

        uint256 priceFloor = mulDivDown(assets, WAD, units);
        uint256 priceCeil  = mulDivUp(assets, WAD, units);

        // Invariant: priceCeil must not exceed the price agreed in take().
        // Violation means a buyer with maxPrice = buyerPrice is wrongfully
        // reverted by Router._execute().
        if (priceCeil > buyerPrice) {
            emit log("=== WRONGFUL TERMINATION CASE ===");
            emit log_named_uint("units",      units);
            emit log_named_uint("assets",     assets);
            emit log_named_uint("buyerPrice", buyerPrice);
            emit log_named_uint("priceFloor", priceFloor);
            emit log_named_uint("priceCeil",  priceCeil);
            emit log_named_uint("remainder",  (assets * WAD) % units);

            assertFalse(true, "priceCeil > buyerPrice: wrongful revert in
            ↪  _execute()");
        }
    }
}
```

Output:

```
=== WRONGFUL TERMINATION CASE ===
units: 2044440078875
assets: 1915433102404
buyerPrice: 936898626766208315
priceFloor: 936898626766312933
priceCeil: 936898626766312934
remainder: 1064547409625
Error: Assertion Failed
```

Issue L-8: Hardcoded `address(this)` Receiver in Withdrawal Functions Exposes Withdrawn Funds to Theft [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/60

Summary

`midnightWithdrawCollateral` and `midnightWithdraw` hardcode `address(this)` as the token receiver when calling the underlying Morpho Midnight protocol. This means withdrawn tokens land in the adapter contract rather than the user's wallet. Since the adapter holds no ownership records and enforces no access control over its token balance, any residual balance is permanently exposed to theft by any subsequent caller.

Vulnerability Detail

Both `midnightWithdrawCollateral` and `midnightWithdraw` invoke the underlying Morpho Midnight functions with `receiver = address(this)`:

```
MORPHO_MIDNIGHT.withdrawCollateral(
    market,
    collateralIndex,
    assets,
    onBehalf,
    address(this)  // tokens land here, not with the user
);

MORPHO_MIDNIGHT.withdraw(market, units, onBehalf, address(this));
```

The underlying protocol intentionally exposes a `receiver` parameter on both functions. The adapter ignores this and hardcodes itself as the receiver, causing all withdrawn tokens to land in the adapter rather than the user's wallet. Since the adapter holds no ownership records and enforces no access control over its token balance, any residual

balance is fully exposed – any address can claim it in a subsequent multicall via `CoreAdapter.erc20Transfer`.

This is not a user mistake. The adapter does not expose a `receiver` argument at all; the choice to send funds to `address(this)` is made entirely by the contract. The issue is also asymmetric with `midnightRepay` and `midnightSupplyCollateral`. In those inbound flows, a missing pre-fund results in a revert and no loss of funds. In contrast, a missing sweep in the withdrawal functions creates a silent and permanent loss condition rather than failing safely.

Impact

Any user whose last bundle action is `midnightWithdrawCollateral` or `midnightWithdraw` without an appended sweep call permanently loses their withdrawn funds. The residual balance is fully exposed to theft by any address in a subsequent multicall. There is no recovery mechanism – the adapter has no owner, no treasury, and no rescue function.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/bundler/MidnightAdapterBase.sol#L92 https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/bundler/MidnightAdapterBase.sol#L105

Tool Used

Manual Review

Recommendation

Expose a `receiver` parameter on both withdrawal functions and forward it to Morpho Midnight instead of hardcoding `address(this)`. That lets the caller decide where the withdrawn tokens should go and removes the unsafe custody step.

Issue L-9: Instant `setAllowlist()` Revocation Breaks Gate Timelock [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/61

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

Summary

`VaultV2AllowlistGate.setAllowlist()` lets the owner instantly revoke any user's gate permissions with no delay or notice. This breaks Morpho Vault V2's documented timelock requirement for gate changes, which exists to give users time to exit before restrictive changes take effect.

Since `withdraw()`, `redeem()`, `transfer()`, and `transferFrom()` all depend on gate permissions, a single owner transaction can immediately freeze any user's ability to exit the vault – with no warning and no recourse.

Vulnerability Detail

Morpho Vault V2's gate architecture is designed around timelocked gate modifications. The documentation states that gate changes are timelocked to preserve the vault's noncustodial guarantees and give users time to react before restrictive changes become active.

However, `VaultV2AllowlistGate.setAllowlist()` allows the owner to modify gate permissions instantly without any delay mechanism.

This is particularly dangerous because the gate controls permissions that directly affect withdrawals and transfers:

- `withdraw()` and `redeem()` require both `canSendShares(onBehalf)` and `canReceiveAssets(receiver)`.
- `transfer()` and `transferFrom()` require both `canSendShares(from)` and `canReceiveShares(to)`.

As documented by Morpho, `SendSharesGate` can lock users out of exiting the vault, while `ReceiveAssetsGate` can prevent users from receiving assets during withdrawals.

As a result, the gate owner can immediately revoke permissions for any address and effectively freeze a user's ability to withdraw, redeem, or transfer vault shares. Unlike the standard Vault V2 gate model, affected users receive no advance notice and have no opportunity to exit before the restriction takes effect.

This breaks the timelock and noncustodial guarantees that Morpho Vault V2 gates are intended to provide.

Impact

The gate owner can immediately and silently freeze any user's exit from the vault by revoking allowlist permissions in a single transaction. This:

- Breaks Morpho Vault V2's noncustodial guarantees by enabling instant, unilateral fund lockout
- Eliminates the user-protection window that the timelock mechanism is designed to provide
- Creates a custodial chokepoint inside an otherwise noncustodial architecture, exposing users to rug-style freezing without any on-chain warning signal

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/gates/VaultV2AllowlistGate.sol#L40-L48

Tools Used

Manual review

Recommendation

Consider implementing a timelocked allowlist update mechanism so permission removals only become effective after a delay, giving users sufficient time to react and

exit if desired.

Issue L-10: Stale `previewWithdraw` Causes Collateral Overdraft on VaultV1 [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/62

Summary

`MidnightWithdrawVaultSharesCallback.onBuy` uses `IERC4626.previewWithdraw` to determine how many vault shares to pull from the buyer's collateral before calling `vault.withdraw`. On VaultV1, `previewWithdraw` does not accrue interest, returning a stale (inflated) share count. `vault.withdraw` accrues interest internally and burns fewer shares than were withdrawn from collateral. The difference is permanently stranded in the callback contract.

Vulnerability Detail

ERC-4626 requires `previewWithdraw` to return the number of shares that would be burned for a given asset amount. Critically, the standard does not require this preview to account for pending interest – implementations that accrue interest lazily will return a stale value until the next on-chain accrual.

VaultV2 handles this correctly. It overrides `previewWithdraw` to call `accrueInterestView()` before computing the share conversion, ensuring the preview always reflects up-to-date share prices:

```
// VaultV2 - previewWithdraw accounts for accrued interest:
function previewWithdraw(uint256 assets) public view returns (uint256) {
    (uint256 newTotalAssets, uint256 performanceFeeShares, uint256
    ↪  managementFeeShares) = accrueInterestView();
    uint256 newTotalSupply = totalSupply + performanceFeeShares +
    ↪  managementFeeShares;
    return assets.mulDivUp(newTotalSupply + virtualShares, newTotalAssets + 1);
}
```

Because `previewWithdraw` and `withdraw` use the same post-accrual state, the share count estimated by the callback matches what the vault actually burns. No surplus arises.

VaultV1 does not override `previewWithdraw`, inheriting the base ERC-4626 implementation which uses stale `totalAssets` with no interest accrual. Its `withdraw`, however, does call `_accrueInterest()` before computing shares (VaultV1 source, L583):

```
// VaultV1 withdraw() - accrues interest before share conversion:
function withdraw(uint256 assets, address receiver, address owner) public override
↪  returns (uint256 shares) {
    _accrueInterest();
    shares = _convertToSharesWithTotals(assets, totalSupply(), lastTotalAssets,
    ↪  Math.Rounding.Ceil);
    _withdraw(_msgSender(), receiver, owner, assets, shares);
}
```

This creates a split: `previewWithdraw` returns a stale, inflated share count `S_stale`, while `withdraw` post-accrual burns only `S_actual < S_stale`. The callback uses `previewWithdraw` to decide how many shares to pull from the buyer's collateral, then hands them all to the vault – but the vault only consumes `S_actual`. The remainder `S_stale - S_actual` is left in the callback contract with no recovery path:

```
// Inflated estimate - pulls too many shares from buyer's collateral:
uint256 sharesToWithdraw =
↪  IERC4626(callbackData.vault).previewWithdraw(buyerAssets);
MORPHO_MIDNIGHT.withdrawCollateral(market, callbackData.collateralIndex,
↪  sharesToWithdraw, buyer, address(this));

// vault.withdraw accrues internally, burns S_actual < sharesToWithdraw:
IERC4626(callbackData.vault).withdraw(buyerAssets, address(this), address(this));
// surplus shares now stranded in address(this) forever
```

The magnitude of the loss scales with time elapsed since the last on-chain accrual for that vault. In low-activity markets this window can span days or weeks, making the loss larger the longer the vault goes without interaction.

Note: Any vault where `withdraw` burns fewer shares than `previewWithdraw` estimated is a fully compliant ERC-4626 implementation. The standard explicitly permits this (EI

P-4626): *"MUST return as close to and no fewer than the exact amount of Vault shares that would be burned in a withdraw call in the same transaction. I.e. withdraw should return the same or fewer shares as previewWithdraw if called in the same transaction."* The callback incorrectly assumes parity between the two. Any compliant vault that accrues interest lazily will silently produce a surplus that is permanently stranded in the contract.

Impact

The buyer loses a portion of their vault share collateral on every `onBuy` execution routed through a VaultV1 vault. The lost shares accrue to the callback contract with no mechanism for recovery. Loss magnitude is proportional to accrued-but-not-yet-settled interest at time of execution. In low-activity markets this will be substantial.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/callbacks/MidnightWithdrawVaultSharesCallback.sol#L46-L52

Tools Used

Manual review

Recommendation

After calling `vault.withdraw`, compare shares burned against shares withdrawn from collateral and re-deposit any surplus back into the buyer's Midnight collateral position:

```
uint256 sharesEstimate = IERC4626(callbackData.vault).previewWithdraw(buyerAssets);

MORPHO_MIDNIGHT.withdrawCollateral(
    market, callbackData.collateralIndex, sharesEstimate, buyer, address(this)
);

uint256 sharesUsed = IERC4626(callbackData.vault).withdraw(
    buyerAssets, address(this), address(this)
```

```
);

uint256 surplus = sharesEstimate - sharesUsed;
if (surplus > 0) {
    IERC4626(callbackData.vault).approve(address(MORPHO_MIDNIGHT), surplus);
    MORPHO_MIDNIGHT.supplyCollateral(
        market, callbackData.collateralIndex, surplus, buyer, new bytes(0)
    );
}
```

Issue L-11: Untrusted `deposit()` Return Value Allows Silent Collateral Loss [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/63

Summary

`MidnightSupplyVaultSharesCallback.onSell` forwards the return value of `IERC4626.deposit()` directly to `supplyCollateral` without verifying that the reported shares match what was actually minted. A vault that returns `0` will cause `supplyCollateral` to be called with an `0` share count. The deposited loan tokens are consumed by the vault, shares are minted to the callback contract, and they are never forwarded as collateral.

Vulnerability Detail

`onSell` deposits loan tokens into the vault and uses the return value of `deposit()` to determine how many shares to supply as collateral:

```
IERC20(loanToken).forceApprove(vault, totalDeposit);
uint256 shares = IERC4626(vault).deposit(totalDeposit, address(this));

IERC20(vault).forceApprove(address(MORPHO_MIDNIGHT), shares);
MORPHO_MIDNIGHT.supplyCollateral(
    market,
    callbackData.collateralIndex,
    shares,
    seller
);
```

If `deposit()` returns `0`, `supplyCollateral` is called with `shares = 0`. Midnight silently no-ops and the minted shares remain stranded in the callback contract with no recovery path.

> The EIP-4626 specification does not require `deposit()` return value to equal shares actually minted. A compliant vault may return zero while minting the correct amount internally.

VaultV1 and VaultV2 currently return the correct share count from `deposit()`, so the issue is not exploitable with presently supported vaults. However, if support is extended to additional ERC-4626 implementations by Midnight/Morpho – which `validateVaultCollateral` does not technically prevent – the assumption breaks silently. The deposited `loanToken` is consumed, shares are minted, and the seller receives no collateral position.

Impact

Permanent loss of user funds. The seller's loan tokens are deposited into the vault and vault shares are minted to the callback contract, but no collateral is supplied to Midnight on the seller's behalf. The seller receives a borrow position with no collateral backing, or the transaction silently no-ops leaving funds unrecoverable. No recovery path exists in the contract.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/callbacks/MidnightSupplyVaultSharesCallback.sol#L65-L68

Tools Used

Manual review

Recommendation

Instead of trusting the return value of `deposit()`, read the actual post-deposit share balance of the callback contract and use that as the amount to forward:

```
IERC20(loanToken).forceApprove(vault, totalDeposit);

uint256 sharesBefore = IERC20(vault).balanceOf(address(this));
IERC4626(vault).deposit(totalDeposit, address(this));
```

```
uint256 shares = IERC20(vault).balanceOf(address(this)) - sharesBefore;

if (shares == 0) revert ZeroSharesMinted();

IERC20(vault).forceApprove(address(MORPHO_MIDNIGHT), shares);
MORPHO_MIDNIGHT.supplyCollateral(
    market,
    callbackData.collateralIndex,
    shares,
    seller
);
```

Issue L-12: `maxLtv` Ceiling Check Provides No Floor, Allowing Excess Collateral on Price Movement [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/64

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

Summary

The `maxLtv` parameter only guards against over-borrowing. It provides no lower bound on collateral efficiency. If the collateral price rises between offer creation and execution, the pro-rata supply may result in an LTV well below `maxLtv`, meaning the borrower posts more collateral than necessary to achieve their intended leverage.

Vulnerability Detail

Collateral amounts are computed pro-rata against `offerSellerAssets` at execution time:

```
uint256 supplyAmount = configAmount.mulDivDown(
    sellerAssets,
    callbackData.offerSellerAssets
);
```

The `maxLtv` check that follows only reverts if the resulting LTV *exceeds* the ceiling:

```
if (accountLtv > callbackData.maxLtv) revert CallbackLib.InvalidLtv();
```

There is no corresponding floor. If collateral price appreciates between offer creation and fill, the same `supplyAmount` covers a higher collateral value, driving LTV below the borrower's target. The take succeeds silently with excess collateral supplied in Midnight.

Impact

The borrower's capital efficiency is reduced – more collateral is posted than their `maxLtv` target requires – but the position remains healthy and withdrawable.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/callbacks/MidnightSupplyCollateralCallback.sol#L53-L74

Tools Used

Manual review

Recommendation

For borrowers who want tighter capital efficiency, consider documenting a `minLtv` parameter as a extension.

Issue L-13: Semantic of `maxSharePriceE27` slippage [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/66

Vulnerability Detail

```
/// @param maxSharePriceE27 Upper bound on the realized per-share price.
```

```
File: MidnightVaultExecutor.sol
41:     function depositAndAddCollateral(
42:         address vault,
43:         Market memory market,
44:         uint256 collateralIndex,
45:         uint256 assets,
46:         uint256 shares,
47:         uint256 maxSharePriceE27,
48:         address onBehalf
49:     ) external returns (uint256 depositedShares, uint256 usedAssets) {
```

For the `maxSharePriceE27`, it is unclear whether this should be "human-normalized" amounts or raw token amounts.

If it means raw token amounts, that is correct, but the NatSpec should explicitly state that it refers to raw token amounts to avoid potential confusion for integrators.

However, if it means "human-normalized" amounts, there will be an issue.

Assume asset = USDC (6 decimals). In this case, the VaultV2's share will be 18 decimals (due to internal decimalOffset)

A user sets maxSharePriceE27 = 1e27, believing it caps the price at 1.0. The user deposits 1000 USDC into the VaultV2.

```
assets_raw = 1000 × 10^6  = 1,000,000,000                    (1000e6)
shares_raw = 1000 × 10^18 = 1,000,000,000,000,000,000,000    (1000e18)
```

```
(USDC vault mints 18-decimal shares; 1:1 in human terms)
```

Then, the `UtilsLib.mulDivUp(usedAssets, RAY, depositedShares)` code translates to:

```
assets_raw × RAY / shares_raw
1000e6 × 1e27 / 1000e18 = 1e15
```

So the check `1e15 > maxSharePriceE27 (1e27)` will only trigger if the user is being overcharged by a factor of 1e12.

Impact

Users might potentially set the wrong slippage due to a misunderstanding.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/c60065ceef92ea2298a2452333625e40083d18ce/MidnightVaultExecutor.sol#L47

Tool Used

Manual Review

Recommendation

Review and update the implementation if the slippage is meant to be ”human-normalized” amounts. Otherwise, consider updating the documentation accordingly.

Issue L-14: `IntentSettler.isRatified()` should be only callable by Midnight [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/67

Vulnerability Detail

Midnight's native EcrecoverRatifier and SetterRatifier both gate `isRatified()` with

```
require(msg.sender == MIDNIGHT, NotMidnight());
```

Since the `IntentSettler.isRatified()` function is intended to be invoked only by Midnight (also mentioned in the comment), it is recommended to implement a similar Midnight-only caller check here for defense-in-depth.

```
File: IntentSettler.sol
65:     /// @notice User-as-maker path. Midnight invokes this when the user is the
 ↪  offer maker.
66:     /// @dev `data` must decode to `(Intent intent, bytes ratifierData)`.
 ↪  `ratifierData` is
67:     ///      forwarded verbatim to the inner ratifier so the taker can pass
 ↪  ratifier-specific
68:     ///      auth/proof material (e.g. signatures, attestations) through.
69:     function isRatified(Offer memory offer, bytes memory data) external view
 ↪  returns (bytes32) {
70:         if (msg.sender != address(MORPHO_MIDNIGHT)) revert NotMidnight();
71:
72:         (Intent memory intent, bytes memory ratifierData) = abi.decode(data,
 ↪  (Intent, bytes));
73:
74:         if (intent.user != offer.maker) revert MakerUserMismatch();
75:         if (!isAuthorized[intent.user][intent.ratifier]) revert
 ↪  RatifierNotAuthorized();
76:
77:         IIntentRatifier(intent.ratifier)
```

```
78:             .onIntentRatify(offer.maker, offer.callback, intent, offer,
↪  offer.callbackData, ratifierData);
79:
80:         return CALLBACK_SUCCESS;
81:     }
```

Impact

An unauthorized address might call `IntentSettler.isRatified()` function, leading to unexpected edge cases.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/fb31acfb4de4d88978cabe757a539000358d0b01/IntentSettler.sol#L69

Tool Used

Manual Review

Recommendation

Ensure that the `IntentSettler.isRatified()` function is only callable by Midnight.

Issue L-15: `sourceMarket` can be the same as `market` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/69

Vulnerability Detail

Instance 1 - `LendMidnightRenewalCallback.onBuy()`

The `sourceMarket` must always be different from the `market`, and this is documented.

```
File: ILendMidnightRenewalCallback.sol
11: interface ILendMidnightRenewalCallback is IBuyCallback {
12:     /// @notice Data passed to the callback when a BUY offer is taken
13:     /// @dev sourceMarket must not be the same as the target market (the one
↪  being bought into).
```

Thus, to avoid edge cases and for defense-in-depth, consider explicitly adding a validation check (`sourceMarket != market`) to ensure that.

```
File: LendMidnightRenewalCallback.sol
31:     function onBuy(
32:         bytes32, /* marketId */
33:         Market memory market,
34:         uint256 buyerAssets,
35:         uint256 units,
36:         uint256, /* pendingFeeIncrease */
37:         address buyer,
38:         bytes memory data
39:     ) external override returns (bytes32) {
40:         if (msg.sender != address(MORPHO_MIDNIGHT)) revert
↪  CallbackLib.OnlyMidnight();
41:         if (buyerAssets == 0 || units == 0) revert CallbackLib.ZeroAmount();
42:
43:         CallbackData memory callbackData = abi.decode(data, (CallbackData));
```

```
44:         if (callbackData.sourceMarket.loanToken != market.loanToken) revert
 ↪  CallbackLib.TokenMismatch();
45:
46:         bytes32 sourceMarketId = IdLib.toId(callbackData.sourceMarket,
 ↪  INITIAL_CHAIN_ID, address(MORPHO_MIDNIGHT));
47:         (uint128 creditAfterUpdate,,) =
48:             MORPHO_MIDNIGHT.updatePositionView(callbackData.sourceMarket,
 ↪  sourceMarketId, buyer);
49:         if (creditAfterUpdate == 0) revert CallbackLib.ZeroAmount();
```

Instance 2 - `BorrowMidnightRenewalCallback.onSell()`

`BorrowMidnightRenewalCallback.onSell()` is the direct mirror of `LendMidnightRenewalCallback.onBuy()`. However, it does not ensure that the `sourceMarketId != marketId`. Thus, the same validation check should be applied.

Impact

Allowing `sourceMarket` to be the same as `market` might lead to unexpected edge cases.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/789d2ea6c7fac11112ebacb244278229d1543e4c/LendMidnightRenewalCallback.sol#L47

Tool Used

Manual Review

Recommendation

Refer to above.

Issue L-16: Maker callback reentrancy bypasses `executeAndConsume` shared cap [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/70

Vulnerability Detail

To recap, the following is the purpose of `consumed`:

> On Midnight, the `consumed[maker][group]` counter is what caps a resting (maker) offer. Every time someone takes the offer, Midnight bumps `consumed[maker][group]` and rejects once it exceeds the offer's `maxAssets/maxUnits`

For some users, they do not care how it gets filled, as long as the G's cap is adhered to. It can be a lender comes and fill the resting order OR the user themselves actively take a lender's order in the market. Either way, the total must not exceed G's cap.

The `executeAndConsume` helps the user to achieve this intention. It does this by manually writing the active leg's fill into the same `consumed[initiator][G]` slot that Midnight uses for the passive leg. Now the resting offer and the active take share one cap:

- Active take fills $100k => router writes `consumed[Alice][G] += 100k`.
- Anyone trying to take her resting offer afterward sees `consumed[Alice][G]` already at the cap and is rejected.

However, there is an edge case where this can be broken. Refer to the following scenario:

Assume that Alice is a borrower in Tenor. She wants to renew her expiring loan but cap her total new exposure at $100k worth of fills, no matter how the fill is sourced.

Alice posts a "resting" SELL offer (group `G`, `maxAssets = $100k`) on the order book, hoping that some lenders will show up and fill her order.

While waiting, Alice saw a good offer and decided to take a lender's BUY offer for the

same $100k. She expects that this spot will fill to eat the shared cap, so the resting order cannot be filled.

Assume that Bob (malicious) is the maker of the BUY offer Alice takes, and his offer has a maker-side callback.

When Alice's batch calls `Midnight.take` on Bob's offer using Router's `executeAndConsume` function, the `take()` function books Alice's first $100K debt. Note that since Alice is a taker here, the consumed check is done on the maker side (`consumed[Bob][Bob's Group]`), not Alice's `consumed[Alice][G]`, so Bob's consumption (Not Alice's) will increment and be checked against.

Then, Bob's callback fires mid-take, before the router has written Alice's consumption. Inside that callback, it calls `Midnight.take` on Alice's still-open resting SELL offer. Midnight checks `consumed[Alice][G]` and sees `0`, and happily fills it for $100k, and incurs a debt of another $100K. Alice's total debt = $200K.

Then, the execution of Alice's outer fill/take resumes and completes. The router explicitly writes `setConsumed(G, 100k + 100k)`, which sets Alice's `consumed` to 200K.

So, the user's initial expectation that it won't exceed the G's cap of 100K is broken in this edge case.

Impact

User's active and resting orders can both fill before consumed is updated, exceeding their intended exposure/debt cap.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/57241410a6316af4b4d7b592b68230abd3f36e6b/TenorRouterAdapterBase.sol#L41

Tool Used

Manual Review

Recommendation

Consider snapshotting `consumed[initiator][consumeGroup]` before execution, then after execution assert it is unchanged before applying the expected fill delta. If it changed during the batch, revert.

Issue L-17: TAKE_ON_BEHALF sentinel resolves against keeper instead of `intent.user` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/71

Vulnerability Detail

Assume that the keeper uses `TAKE_ON_BEHALF` with `type(uint256).max` sentinel. In this case, the `TenorRouterAdapterBase._executeResolvingSentinels()` resolves the `type(uint256).max` sentinel for `maxFill/minFill` using `_initiator()`. The `_initiator()` points to the Bundler3 initiator (the keeper)

```
File: TenorRouterAdapterBase.sol
57:     function _executeResolvingSentinels(ExecuteParams calldata params, Action[]
↪  calldata actions)
..SNIP..
64:         if (actions.length > 0 && (maxFill == type(uint256).max || minFill ==
↪  type(uint256).max)) {
65:             address initiator = _initiator();
66:             uint8 fillIndex = _fillIndex(params.fillAxis, actions, initiator);
67:             uint256 resolved = _resolveSentinel(fillIndex, initiator, actions);
68:             if (resolved == 0) revert SentinelResolvedToZero(fillIndex);
69:             if (maxFill == type(uint256).max) maxFill = resolved;
70:             if (minFill == type(uint256).max) minFill = resolved;
71:         }
```

and `_resolveSentinel` reads that address's position in `actions[0].offer.market`. So, in Line 88, it will end up reading the keeper's debt.

```
File: TenorRouterAdapterBase.sol
80:     function _resolveSentinel(uint8 fillIndex, address initiator, Action[]
↪  calldata actions)
81:         internal
```

```
82:         view
83:         returns (uint256)
84:     {
85:         if (fillIndex == RouterLib.FILL_UNITS) {
86:             bytes32 id = IdLib.toId(actions[0].offer.market, _INITIAL_CHAIN_ID,
 ↪  address(_MORPHO_MIDNIGHT));
87:             if (_isBuyerSide(actions[0], initiator)) {
88:                 return _MORPHO_MIDNIGHT.debtOf(id, initiator);
89:             }
90:             if (_MORPHO_MIDNIGHT.tickSpacing(id) == 0) return 0;
91:             (uint128 creditAfterUpdate,,) =
 ↪  _MORPHO_MIDNIGHT.updatePositionView(actions[0].offer.market, id, initiator);
92:             return creditAfterUpdate;
```

For `MIDNIGHT_TAKE` and first-party `TAKE_ON_BEHALF` (`offer.maker == initiator`), the `initiator` is a direct party to the trade, so this is the right account. However, for third-party `TAKE_ON_BEHALF` (`offer.maker != initiator`, i.e. keeper-relayed), the taker recorded on Midnight is `intent.user`, not the initiator:

```
File: IntentSettler.sol
40:     function take(
..SNIP..
60:         (buyerAssets, sellerAssets) = MORPHO_MIDNIGHT.take(
61:             offerData.offer, takeUnits, intent.user, receiver, takerCallback,
 ↪  takerCallbackData, offerData.ratifierData
62:         );
```

So the sentinel never consults the execution mode, which resolves against the keeper's position instead of `intent.user`'s.

Two resulting behaviors when a keeper uses the sentinel with `TAKE_ON_BEHALF` mode:

- Keeper holds no position in `actions[0].offer.market` => resolves to `0` => `revert SentinelResolvedToZero`.
- Keeper holds an unrelated position there => resolves to the keeper's debt/credit => `maxFill/minFill` silently sized to the wrong amount.

Consider the following scenario:

- Alice has a borrow on Midnight market `M`: debt = 100,000 units.
- Keeper `K` runs Alice's pre-authorized exit/close-out as a single third-party `TAKE_ON_BEHALF` action on `M` (`intent.user = Alice`, `offer.maker != K`), on the `FILL_UNITS` axis, with `maxFill = minFill = type(uint256).max`.

```
offer.buy = false
offer.maker = lender / counterparty
intent.user = Alice
takerCallback = BorrowMidnightToBlueCallback
```

Then:

1. `Bundler3.multicall` records `initiator() = K`.
2. `_executeResolvingSentinels`: `maxFill == type(uint256).max` => resolution branch, `initiator = K`.
3. `_resolveSentinel(FILL_UNITS, K, actions)`:
    - `debtOf(M, K) = 0` (K has no position in `M`),
    - `tickSpacing(M) > 0` (live market),
    - `updatePositionView(M, id, K) = 0`,
    - returns `0`.
4. `resolved == 0` => `revert SentinelResolvedToZero(FILL_UNITS)`.

Alice has a valid 100,000 debt to close, but the resolution is performed against K, which leads to a revert at the end.

In addition, if the keeper happens to hold debt in `M`, then it will resolve to the keeper's debt number and silently proceed, leading to silent "mis-sizing".

Impact

Keeper-relayed close-outs can revert or silently mis-size fills, making valid user migrations unreliable when using `type(uint256).max` bounds

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/57241410a6316af4b4d7b592b68230abd3f36e6b/TenorRouterAdapterBase.sol#L88

Tool Used

Manual Review

Recommendation

In the current design, the `TAKE_ON_BEHALF` with `type(uint256).max` sentinel does not work. Thus, it should revert with an explicit error if such a configuration is encountered and/or document that the `maxFill`/`minFill` sentinel is valid only when the initiator is the position owner (first-party), and keepers must pass concrete bounds instead.

Issue L-18: Opening migrations silently net against existing opposite-side target positions [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/73

Vulnerability Detail

In a Midnight market, an account can be on only one side at a time.

A take nets against the opposite leg before opening a new one. So, a seller's existing credit is consumed first, and only the overflow becomes new debt (the buyer's case is symmetric).

```
File: Midnight.sol
376:         uint256 buyerCreditIncrease = UtilsLib.zeroFloorSub(units,
↪  buyerPos.debt);
377:         uint256 sellerCreditDecrease = UtilsLib.min(units, sellerPos.credit);
378:         uint256 sellerDebtIncrease = units - sellerCreditDecrease;
```

For the four opening callbacks (`BorrowBlueToMidnightCallback`, `BorrowMidnightRenewalCallback`, `LendVaultToMidnightCallback`, `LendMidnightRenewalCallback`), the issue is that the user already holds an opposite side position in the target market, and the fill lands within it and the take nets that position away.

Assume using the `BorrowBlueToMidnight` and 1-year maturity.

- Alice holds an existing 10,000-unit Midnight lend claim (e.g., worth 9,524) in the target Midnight market
- Alice owes 9,800 USDC debt on Blue

Alice intends to migrate her debt position from Blue to Midnight. Thus, she authorized the `IntentSettler` and configured the necessary migration parameters.

Bob, the keeper, calls the `Midnight.take` function with `offer` parameter set to `maker = Al`

`ice`, `ratifier = IntentSettler`, `buy = false` (SELL), `callback = BorrowBlueToMidnightCallback`, and `taker` parameter set to `Bob`.

The issue here is that Midnight nets the existing 10,000 credit away from Alice's account, and the remaining 290 will be counted as debt. The user ”receives” the proceeds, and the callback spends them repaying the Blue debt. Notice that Alice's prior target-market lend claim has been netted away rather than remaining as a separate position.

Alice might expect that the end state to have two separate positions, such as follows:

- 10,000-unit Midnight lend claim
- 10,290-debt in Midnight (migrated from Blue)

However, due to how Midnight is designed, this is not possible as credit and debt are shared and net/offset during operation.

Impact

Users can lose/consume existing target-market credit or debt during migration instead of ending with a separate new position, causing unexpected position changes.

Code Snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/e40450dbda3d199602f28c920da0c8441332ec10/BaseMigrationRatifier.sol#L104

Tool Used

Manual Review

Recommendation

If this behavior is expected per the design, it is recommended to document it somewhere. Otherwise, the code might need to block any migration where the user already holds an opposite-side position in the target market (no existing credit for the borrow-opening callbacks; no existing debt for the lend-opening callbacks).

Issue L-19: `_maxUnitsInterest()` defaults to `remainingBudget` on zero price [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/74

Summary

CallbackFeeAdjuster's `beforeDispatch()` -> `_maxUnitsInterest()` route returns current budget back instead of zero and max number in the zero and infinite price cases. This is grossly incorrect with regard to CallbackFeeAdjuster's goals, i.e. when price is zero a seller doesn't want to sell anything (as seller receives zero amount now no matter what future debt is), while buyer wants to buy max, and vice versa for the infinite price case. Now there is neither revert happens nor zero or max quantity being returned, while defaulting to `remainingBudget` that happens is incorrect in all such cases.

Vulnerability Detail

When `netSellerPrice == 0` logic falls back to `return remainingBudget` (`CallbackFeeAdjuster.sol:128, :158`). `netSellerPrice` is `0` when Midnight's `tradingFee >= offerPrice`, and in this case the seller-taker gets `sellerAssets = 0` for any fill, so the correct cap is `0` ("take none"), not the whole remaining budget. Similarly, it is `type(uint128).max` ("take all capacity of the offer") for buyer-taker.

Other size logics already returns `0` for the seller case (`SupplyCollateralCallbackClamp`, `LendMidnightRenewalClamp`) and `type(uint128).max` for the buyer case (`BuyOfferClamp`, `VaultWithdrawClamp`), `CallbackFeeAdjuster` being the outlier.

Impact

Executable window looks to be `0 < offerPrice == tradingFee` (strict < underflows in Midnight's `take()`), while adjuster is only called when fees are active.

This way, when `offerPrice == tradingFee` the effective price is zeroed out and `_capTakeUnits()` returns the whole remaining offer capacity, making `_execute()` -> `_dispatch()` -> `_`

`dispatchTakeOnBehalf()` and `_dispatchMidnightTake()` go for the full take, which is loss making for the user in this case.

Code Snippet

CallbackFeeAdjuster.sol#L125-L128

```
uint256 price = fillIndex == RouterLib.FILL_BUYER_ASSETS
    ? RouterLib.netBuyerPrice(offerPrice, settlementFee, feeRate)
    : RouterLib.netSellerPrice(offerPrice, settlementFee, feeRate);
if (price == 0) return remainingBudget;
```

CallbackFeeAdjuster.sol#L136-L158

```
function _maxUnitsPercentage(Offer calldata offer, uint8 fillIndex, uint256
↪  remainingBudget, uint256 feeRate)
    internal
    view
    returns (uint256)
{
    uint256 offerPrice = TickLib.tickToPrice(offer.tick);
    bytes32 marketId = IdLib.toId(offer.market, INITIAL_CHAIN_ID,
    ↪  address(MORPHO_MIDNIGHT));
    uint256 secondsToMaturity = UtilsLib.zeroFloorSub(offer.market.maturity,
    ↪  block.timestamp);
    uint256 settlementFee = MORPHO_MIDNIGHT.settlementFee(marketId,
    ↪  secondsToMaturity);

    if (fillIndex == RouterLib.FILL_BUYER_ASSETS) {
        // raw + floor(raw*feeRate/WAD) = floor(raw*(WAD+feeRate)/WAD) for integer
        ↪  raw, so the
        // tight bound is rawMax = max{r : floor(r*(WAD+feeRate)/WAD)   R}.
        uint256 buyerPriceMidnight = offerPrice + settlementFee;
        if (buyerPriceMidnight == 0) return remainingBudget;
        uint256 rawMax = (remainingBudget + 1).mulDivUp(WAD, WAD + feeRate) - 1;
        return rawMax.mulDivDown(WAD, buyerPriceMidnight);
    }
```

```
    // raw - floor(raw*feeRate/WAD) = ceil(raw*(WAD-feeRate)/WAD) for integer raw.
    // feeRate   MAX_PERCENTAGE_FEE_RATE < WAD, so WAD - feeRate > 0.
    uint256 sellerPriceMidnight = UtilsLib.zeroFloorSub(offerPrice, settlementFee);
    if (sellerPriceMidnight == 0) return remainingBudget;
```

Tool Used

Manual Review

Recommendation

Consider returning `0` (seller) / `type(uint128).max` (buyer) for zero net price (and vice versa for the infinite price) across the max units logic.

Issue L-20: Round trip configuration exposes users to full balance drain griefing [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/75

Summary

When `renewalWindow  minDuration` and a round-trip exists in user configuration, a permissionless caller can convert the full balance of such a user into protocol fees over a course of minutes (depending on fee itself), either as pure DoS or as direct extraction when `feeRecipient` is affiliated with the caller.

Vulnerability Detail

For example, a lender can configure two migration directions via the `IntentSettler`:

- `LendVaultToMidnightCallback.onBuy` migrates vault shares into Midnight credit. It pulls `buyerAssets + fee` from the lender's vault:

```
uint256 sharesBurned = IERC4626(callbackData.vault).withdraw(buyerAssets + fee,
↪  address(this), buyer);
```

- `LendMidnightToVaultCallback.onSell` migrates Midnight credit back into vault shares. It deposits `sellerAssets - fee` into the lender's vault:

```
uint256 depositAmount = sellerAssets - fee;
IERC20(market.loanToken).forceApprove(callbackData.vault, depositAmount);
uint256 shares = IERC4626(callbackData.vault).deposit(depositAmount, seller);
```

Having both ways setup at the same time can be plausible as lenders might be agnostic for the exact vault (from a chosen list) to keep the funds in, focusing on rates mostly, and rate curves can overlap.

`IntentSettler.take()` is permissionless with regard to a caller (`msg.sender == onBehalf` path), any address can invoke it on behalf of a lender whose setup is in place (`Midnight.i`

sAuthorized[lender][Settler] = true, IntentSettler.isAuthorized[lender][ratifier] = true, and userParams for both directions on the same (vaultId, MnTenorMarketId) pair). The V→Mn cadence check at BaseMigrationRatifier._ratifyWindow uses FourWeekCadence.nearestBoundary(t) = (t / 28 days) * 28 days, so the check if (renewalPeriodStart > block.timestamp) revert doesn't fail after the first 28-day boundary past inception.

Given such authorizations an attacker can run the round-trip in a loop. Effect per cycle, on the same units:

Leg	Vault delta	Tenor fee to feeRecipient
V→Mn	− units × buyerPrice × (1 + feeRate)	feeRate × buyerAssets
Mn→V	+ units × sellerPrice × (1 − feeRate)	feeRate × sellerAssets

The Mn→V leg deposits the proceeds back into the vault, so the next cycle's amount is the full new balance, every cycle's fee is taken on the full current balance. After N cycles it's account_N account_0 × ((1 − feeRate) / (1 + feeRate))^N. The approximate asymptote:

feeRate	N to 1% remaining	Time on Ethereum
1% per leg	228	~2–3 min
0.5%	458	~4–7 min
0 (only tradingFee = 5 bp)	~4 600	~35–70 min

Under the misconfig params.renewalWindow params.minDuration, the Mn→V window check block.timestamp < sourceMaturity − renewalWindow is satisfied on the just-created Mn position (since _validateTargetMaturity forces targetMaturity now + minDuration).

Otherwise, loop_period ceil28d(minDuration) − renewalWindow waiting time is enforced.

Impact

User's balance exposed for the configuration can be drain materially fully (to a dust amount) over course of minutes / hours, while attacking caller pays only gas.

Code Snippet

No user configuration check is performed on setting:

MigrationIntentRatifier.sol#L48-L63

```
/// @notice Sets `onBehalf`'s migration params for a `(callback, src, tgt)` tuple.
/// @dev Callable by `onBehalf` itself or anyone they have authorized on midnight.
 ↪  Keeping
///      onBehalf here lets bundler flows atomically update params alongside a take.
function setParams(
    address onBehalf,
    address callback,
    bytes32 sourceTenorMarketId,
    bytes32 targetTenorMarketId,
    UserMigrationParams calldata params
) external override {
    if (msg.sender != onBehalf && !MORPHO_MIDNIGHT.isAuthorized(onBehalf,
     ↪  msg.sender)) {
        revert Unauthorized();
    }
    userParams[onBehalf][callback][sourceTenorMarketId][targetTenorMarketId] =
     ↪  params;
    emit ParamsSet(onBehalf, callback, sourceTenorMarketId, targetTenorMarketId,
     ↪  params);
}
```

IntentSettler's `take()` allows anyone to run a take as long as taker's settings allow the path:

IntentSettler.sol#L40-L63

```
    function take(
        ...
```

```
    ) external returns (uint256 buyerAssets, uint256 sellerAssets) {
>>      if (msg.sender != onBehalf && !MORPHO_MIDNIGHT.isAuthorized(onBehalf,
↪   msg.sender)) {
            revert Unauthorized();
        }
        if (!isAuthorized[intent.user][intent.ratifier]) revert
        ↪  RatifierNotAuthorized();

        MORPHO_MIDNIGHT.touchMarket(offerData.offer.market);

        IIntentRatifier(intent.ratifier)
>>          .onIntentRatify(onBehalf, takerCallback, intent, offerData.offer,
↪   takerCallbackData, ratifierData);

        (buyerAssets, sellerAssets) = MORPHO_MIDNIGHT.take(
            offerData.offer, offerData.ratifierData, takeUnits, intent.user,
            ↪  receiver, takerCallback, takerCallbackData
        );
    }
```

As in `take()` case `onBehalf` is passed as `caller` to `_ratifyRate()` -> `getRate()` and end up not being controlled there, i.e. the `take()` its factually permissionless:

BaseMigrationRatifier.sol#L272-L296

```
function _ratifyRate(
    ...
) internal view {
    uint256 duration = _computeDuration(takerCallback, sourceMaturity,
    ↪  targetMaturity);
    uint256 policyRate = IInterestRatePolicy(params.interestRatePolicy)
        .getRate(
            sourceTenorMarketId,
            targetTenorMarketId,
            renewalPeriodStart,
            caller,
            user,
            sourceMaturity,
            targetMaturity,
```

```
            offer.buy
        );
```

MarketMakingPolicy.sol#L62-L71

```
function getRate(
    bytes32 sourceTenorMarketId,
    bytes32 targetTenorMarketId,
    uint256, /* renewalPeriodStart */
    address, /* caller */
    address taker,
    uint256 sourceMaturity,
    uint256 targetMaturity,
    bool offerIsBuy
) public view virtual returns (uint256) {
```

Tool Used

Manual Review

Recommendation

Consider rejecting `renewalWindow` < `minDuration` in `MigrationIntentRatifier.setParams`. A `maxFeeBudget` per `(user, callback)` decremented per call and reset at cadence boundaries can control fee exposure regardless of loop frequency and exact user callback parameters used.

Issue L-21: `BorrowMidnightToBlueCallback`, `BorrowBlueToMidnightCallback`, `BorrowMidnightRenewalCallback` lack a user-specified post migration `maxLtv` bound, so a permissionless caller can push the migrating borrower to the target market's LLTV [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/76

Summary

Given that, e.g. the target blue market can have a lower LLTV then source market, and there is `callbackData.feeRate` defined fee present the rolling borrower can be placed into a borderline liquidatable position without their consent as IntentSettler's `take()` allows an arbitrary caller to supply `takeUnits`, the size of the `take()`, up to max current debt. This can reduce the distance between user's LTV and market LLTV close to zero contrary to user's intent.

Vulnerability Detail

All three borrower-side callbacks invoked through the `IntentSettler.take()` permissionless keeper path, `BorrowMidnightToBlueCallback.onBuy` (Midnight -> Blue), `BorrowBlueToMidnightCallback.onSell` (Blue -> Midnight), and `BorrowMidnightRenewalCallback.onSell` (Midnight -> Midnight), migrate the borrower's existing position to a target market by pro-rata collateral transfer plus a fresh borrow/take on the target market. The post-migration target LTV is determined entirely by the offer's `price` (`tickToPrice(offer.tick)`), the `callbackData.feeRate`, and the borrower's source LTV at the moment of the migration.

Caller/keeper can impact both `price` (selected via the keeper's offer choice, bounded

only by the borrower's `limitRatePerSecond` floor/ceiling) and the borrower's source LTV (by choosing the moment within the borrower's window). For the two seller callbacks the keeper additionally has the target Midnight market's maturity as a choice, as borrower's `(callback, src, tgt)` authorization covers a family of Midnight markets sharing loanToken/collateral/rcf/gates but with any maturity in the borrower's `[minDur, maxDur]` window.

I.e. the only protection against an over-leveraged target position is each target market's own `LLTV` (enforced by `MORPHO_BLUE.borrow`'s `_isHealthy`, and by Midnight's post-take seller-health check). The borrower has no direct way to enforce a stricter ceiling, their only indirect bound being `limitRatePerSecond`, which doesn't provide LTV control. So the keeper is free to drive the migration up to the target market's `LLTV` ceiling and (when `feeRate > 0`) extract the maximum fee allowed within the borrower's rate band.

Impact

The migrating borrower can be placed at the maximum target LTV the keeper can construct within the borrower's authorized rate band, e.g. right at the target market's `LLTV` boundary, across three distinct migration flows: Midnight -> Blue, Blue -> Midnight, and Midnight -> Midnight. No user-side consent mechanism for a tighter LTV ceiling exists for any of the three. Combined with `callbackData.feeRate > 0`, the keeper additionally maximizes the protocol fee charged on each migration. Subsequent oracle movement that the borrower would have tolerated at their intended (lower) LTV can be enough to trigger liquidation on the target market.

Code Snippet

As an example, in `BorrowMidnightToBlueCallback` case, the fee is added to the Blue borrow amount without any matching collateral, so any `feeRate > 0` strictly raises post-migration Blue LTV:

BorrowMidnightToBlueCallback.sol#L75-L75:

```
uint256 borrowAmount = buyerAssets + fee;
```

`BorrowMidnightToBlueCallback.onBuy` borrows `buyerAssets + fee` from Blue on the borrower's behalf with no post-borrow LTV check, so the resulting Blue LTV is bounded

only by Blue's own `LLTV`:

BorrowMidnightToBlueCallback.sol#L76-L76

```
MORPHO_BLUE.borrow(callbackData.targetMarketParams, borrowAmount, 0, buyer,
↪  address(this));
```

Tool Used

Manual Review

Recommendation

Consider introducing a `maxLVT` parameter to `BorrowMidnightToBlueCallback`, `BorrowBlueToMidnightCallback` and `BorrowMidnightRenewalCallback`, e.g. similarly to `MidnightSupplyCollateralCallback`.

Issue L-22: TenorRouterAdapterBase's `_resolveSentinel()` is based on balance reading for `FILL_BUYER_ASSETS`, so bundles with at least one such element with max `minFill` can be denied by front running [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/77

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

Summary

As `totals[fillIndex]` is bounded by user's intent, when `minFill` is set from adapter's balance reading there is a direct griefing vector: front running with directly sending dust to adapter's balance.

Vulnerability Detail

The "spend all my balance" path requires `cost(takeUnits) == balance`, where `takeUnits` is fixed, while balance can be topped up with a dust amount. A donation can grief bundles with at least one `FILL_BUYER_ASSETS` with `minFill == type(uint256).max` element. This will deny `execute()`, `executeAndConsume()` -> `_executeResolvingSentinels()` -> `_execute()` path, reverting the bundle.

Impact

Execution of any bundle with `FILL_BUYER_ASSETS` and `minFill == type(uint256).max` can be reverted by front running with dust donation.

Code Snippet

`minFill` is based on the current balance:

TenorRouterAdapterBase.sol#L80-L97

```
    function _resolveSentinel(uint8 fillIndex, address initiator, Action[] calldata
    ↪  actions)
        internal
        view
        returns (uint256)
    {
        if (fillIndex == RouterLib.FILL_UNITS) {
            bytes32 id = IdLib.toId(actions[0].offer.market, _INITIAL_CHAIN_ID,
            ↪  address(_MORPHO_MIDNIGHT));
            if (_isBuyerSide(actions[0], initiator)) {
                return _MORPHO_MIDNIGHT.debtOf(id, initiator);
            }
            if (_MORPHO_MIDNIGHT.tickSpacing(id) == 0) return 0;
            (uint128 creditAfterUpdate,,) =
            ↪  _MORPHO_MIDNIGHT.updatePositionView(actions[0].offer.market, id,
            ↪  initiator);
            return creditAfterUpdate;
        } else if (fillIndex == RouterLib.FILL_BUYER_ASSETS) {
>>          return
↪   IERC20(actions[0].offer.market.loanToken).balanceOf(address(this));
        } else {
            revert SentinelNotSupported(fillIndex);
        }
```

Which can't be met when it changes:

TenorRouter.sol#L253-L255

```
if (totals[fillIndex] < minFill) {
    revert InsufficientFill(totals[fillIndex], minFill);
}
```

Tool Used

Manual Review

Recommendation

Consider documenting this sentinel limitation.

Issue L-23: `_dispatchMidnightTake` and flash loan adapter grant unlimited allowance to Midnight, exposing full router balance under compromise [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/79

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

Summary

In two separate places the router grants `type(uint256).max` approval to Midnight – once transiently in `_dispatchMidnightTake` and once in `MidnightAdapterBase` (`midnightFlashLoan`). In both cases only a bounded amount is actually needed. A compromised Midnight can drain the router's full token balance through either path.

Vulnerability detail

Both paths share the same root issue: approval is not scoped to the actual transfer amount.

In `_dispatchMidnightTake`, the exact lender outflow is known before the call, so `type(uint256).max` is unnecessary – Midnight pulls a fixed amount directly with no inner calls.

In `midnightFlashLoan`, max allowance was intentionally chosen to avoid inner calls consuming the approval mid-execution and reverting. The tradeoff is accepted, but the exposure is the same: any token routed through a flash loan has its full balance at risk during the window.

Impact

Under a Midnight compromise, an attacker can drain the router's full balance of any approved token through either path rather than being limited to the current fill or loan

amount. No impact under normal operation.

Code snippet

https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/periphery/TenorRouter.sol#L351-L353 https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/blob/013a304dab2579661cb9aaf5922c21d171bb4259/tenor-morpho-v2-contracts-2/src/bundler/MidnightAdapterBase.sol#L122-L124

Tools Used

Manual review

Recommendation

For `_dispatchMidnightTake`, approve only the exact lender outflow and clear afterward. For the flash loan adapter, consider whether the inner-call revert risk can be mitigated another way – e.g. tracking and restoring allowance state – so that exact amounts can be used there too.

Issue L-24: DelayedLiquidationGate's and MidnightVaultExecutor's `onLiquidate()` allows for stealing these contracts' balances [RESOLVED]

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/80

Summary

In both cases `onLiquidate()` assumes that it was set to receiver in Midnight's `liquidate()`, so it either sends the specified amount of collateral assets directly to `liquidator` or redeems the specified amount of Vault's shares to it. There is no control that it was called for a market tied to the contracts, only Midnight address itself is verified, so an attacker can call it from a liquidation for any other Midnight's market. Attacker can set `receiver` for such a call to be themselves, while callback being DelayedLiquidationGate or MidnightVaultExecutor.

As Midnight is permissionless about market creation, the attacker can construct a market with this collateral and this state of the borrower to be liquidated freely for any valuable token that happens to be on the DelayedLiquidationGate or MidnightVaultExecutor balance. I.e. for any such token a market can be constructed to have it as collateral and a liquidatable borrower can be staged to have the amount equal to contract's balance as collateral / vault shares seized. This can be done on the fly, stealing any valuable balances from these contracts as they appear.

Vulnerability Detail

I.e. in both cases `onLiquidate()` can be used to drain any DelayedLiquidationGate balances of other market collaterals (which means any token as market can be created for it) via supplying it as the callback in a direct `MORPHO_MIDNIGHT.liquidate()` call for a market which liquidation gate allows direct calls, and supplying receiver to be the attacker, while `market.collateralParams[collateralIndex].token` being that collateral

token to sweep with `seizedAssets` chosen to match DelayedLiquidationGate's balance, and `market.loanToken` allowance for `repaidUnits` be in place.

Similarly, in MidnightVaultExecutor's case, any leftover balance of `IERC4626(vault)` token can be swept by anyone via doing a liquidation in a pre-made market with collateral equal to these `IERC4626(vault)` token and `receiver` supplied on liquidation being attacker instead of this contract and `seizedShares` chosen to match the leftover balance.

Impact

DelayedLiquidationGate and MidnightVaultExecutor leftover balances can be stolen fully.

Code Snippet

DelayedLiquidationGate's `onLiquidate()` decodes `liquidator` from caller supplied data and sends `seizedAssets` of `market.collateralParams[collateralIndex].token` there:

DelayedLiquidationGate.sol#L98-L116

```
    function onLiquidate(
        address, /* callerFromMidnight */
        bytes32 marketId,
        ...
    ) external returns (bytes32) {
        if (msg.sender != address(MORPHO_MIDNIGHT)) revert NotMorpho();

>>      (address liquidator, bytes memory liquidatorData) = abi.decode(data,
↪   (address, bytes));

        if (seizedAssets > 0) {
>>
↪   SafeTransferLib.safeTransfer(market.collateralParams[collateralIndex].token,
↪   liquidator, seizedAssets);
        }
```

MidnightVaultExecutor's `onLiquidate()` also decodes `liquidator` from caller supplied

data and redeems `seizedShares` shares of `IERC4626(vault)` to it as receiver:

MidnightVaultExecutor.sol#L205-L222

```
    function onLiquidate(
        address, /* callerFromMidnight */
        bytes32 id,
        Market memory market,
        ...
    ) external override returns (bytes32) {
        if (msg.sender != address(MORPHO_MIDNIGHT)) revert
        ↪  CallbackLib.OnlyMidnight();
>>      (address vault, address liquidator, uint256 minSharePriceE27, bytes memory
↪   callbackData) =
            abi.decode(data, (address, address, uint256, bytes));
        CallbackLib.validateVaultCollateral(market, vault, market.loanToken,
        ↪  collateralIndex);

>>      uint256 redeemed = IERC4626(vault).redeem(seizedShares, liquidator,
↪   address(this));
```

In Midnight `liquidate()` caller can independently supply `receiver` to send the collateral to and `callback`, whose `onLiquidate()` to be called:

Midnight.sol#L696-L717

```
>>
↪   SafeTransferLib.safeTransfer(market.collateralParams[collateralIndex].token,
↪   receiver, seizedAssets);

        if (callback != address(0)) {
            require(
>>              ILiquidateCallback(callback)
                    .onLiquidate(
                        msg.sender,
                        id,
                        market,
                        collateralIndex,
                        seizedAssets,
```

```
                    repaidUnits,
                    borrower,
                    receiver,
                    data,
                    badDebt
                ) == CALLBACK_SUCCESS,
            WrongLiquidateCallbackReturnValue()
        );
    }

    SafeTransferLib.safeTransferFrom(market.loanToken, payer, address(this),
    ↪  repaidUnits);
```

Tool Used

Manual Review

Recommendation

Consider adding `if (receiver != address(this)) revert ...` or `if (callerFromMidnight != address(this)) revert ...` kind of control as a mitigation (an alternative: transient-storage flag set in `liquidate()`, checked in `onLiquidate()`).

Issue L-25: Vault <-> Midnight callbacks allow to steal from their Tenor user via executing dust sized trades

Source: https://github.com/sherlock-audit/2026-05-tenor-markets-may-21st-2026/issues/81

Summary

There can be a combination of high share price and high-value low-decimal underlying (e.g. old vaults with mainnet WBTC underlying). In this case IntentSettler's `take()` caller can profit off taker (or simply grief them) via Vault <-> Midnight callbacks by either burning costly shares for small amount withdrawal (exploiting rounding up) or receiving less shares for small amount deposit (rounding down) by repeating dust amount takes many times. Another vulnerable setup is maker using MidnightSupplyVaultSharesCallback or MidnightWithdrawVaultSharesCallback.

Vulnerability Detail

Attacker can profit off taker losing value to Vault as the receivers of the rounding proceeds are all the other Vault depositors and attacker can become a large depositor before and withdraw after the attack. No special settings on the taker part are required, as one direction allowance for taker's current position is enough, and that's the basic setup all Tenor users shall have.

Affected cases are Vault <-> Midnight callbacks via IntentSettler's `take()` where taker is the victim and also Midnight supply and withdraw Vault shares callbacks where maker is the victim (with direct Midnight takes instead of permissionless IntentSettler's `take()`), i.e. all the flows where third party can invoke dust deposit or withdrawal by the either migration approved taker or Tenor callback using maker.

Together these cases weaponize dust manipulation: it's possible in Midnight without Tenor, but not directly exploitable by itself (one can zero out fees this way, but, given fee percentage being low enough, it doesn't look profitable), but when it's paired with

automatic deposit/withdrawal it can become incentivized for Vault depositors to make Tenor user do so in dust amounts to bleed value via Vault standard against-user rounding direction.

Impact

Net effect can be positive for attacker and material on L2s/L1s with cheap gas, and this gas cost with maybe Vault entry/exit fees (in general case when there can be any) is total cost, which is removed from attacker's profit from taker loss socialization among Vault depositors, creating direct incentives beyond the pure griefing case (when attacker pays gas, receives nothing, taker lose value more or less than gas costs).

For example, when vault shares have same decimals as underlying, WBTC on mainnet, and `2.0` share price, i.e. `1` share is `2` underlyings, cumulative depositors' profit from rounding is around `5x` of L2 (Base) gas cost. This is a slow bleeding due to vast number of transactions required, with daily loss less than `USD 1k` for a given user.

I.e. while max is limited by user's position and only a round trip enabling setup allows to go beyond that (#75), the minimum isn't controlled and no special setup is required to exploit this, only price/decimals/value combination is needed on the Vault level.

Code Snippet

`takeUnits` amount is not validated by itself and is directed to Midnight's `take()` as is, factually having no minimum boundary:

IntentSettler.sol#L40-L63

```
    function take(
        OfferData calldata offerData,
>>      uint256 takeUnits,
        address onBehalf,
        address receiver,
        address takerCallback,
        bytes calldata takerCallbackData,
        Intent calldata intent,
        bytes calldata ratifierData
    ) external returns (uint256 buyerAssets, uint256 sellerAssets) {
```

```
        if (msg.sender != onBehalf && !MORPHO_MIDNIGHT.isAuthorized(onBehalf,
        ↪  msg.sender)) {
            revert Unauthorized();
        }
        if (!isAuthorized[intent.user][intent.ratifier]) revert
        ↪  RatifierNotAuthorized();

        MORPHO_MIDNIGHT.touchMarket(offerData.offer.market);

        IIntentRatifier(intent.ratifier)
            .onIntentRatify(onBehalf, takerCallback, intent, offerData.offer,
            ↪  takerCallbackData, ratifierData);

        (buyerAssets, sellerAssets) = MORPHO_MIDNIGHT.take(
>>          offerData.offer, offerData.ratifierData, takeUnits, intent.user,
↪  receiver, takerCallback, takerCallbackData
        );
    }
```

With attacker being Midnight taker instead of IntentSettler's `take()` caller, with no min amount MidnightSupplyVaultSharesCallback's `onSell()` is also prone to multiple dust deposits manipulation in the costly shares and valuable underlying vault case:

MidnightSupplyVaultSharesCallback.sol#L62-L68

```
uint256 totalDeposit = sellerAssets + amountFromSeller;

IERC20(loanToken).forceApprove(vault, totalDeposit);
uint256 shares = IERC4626(vault).deposit(totalDeposit, address(this));

IERC20(vault).forceApprove(address(MORPHO_MIDNIGHT), shares);
MORPHO_MIDNIGHT.supplyCollateral(market, callbackData.collateralIndex, shares,
↪  seller);
```

With no min `buyerAssets` amount MidnightWithdrawVaultSharesCallback's `onBuy()` is prone to the same manipulation:

MidnightWithdrawVaultSharesCallback.sol#L46-L50

```
uint256 sharesToWithdraw =
↪  IERC4626(callbackData.vault).previewWithdraw(buyerAssets);

MORPHO_MIDNIGHT.withdrawCollateral(market, callbackData.collateralIndex,
↪  sharesToWithdraw, buyer, address(this));

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

Tool Used

Manual Review

Recommendation

Consider introducing minimum amounts for `sellerAssets` and `buyerAssets` in all the callbacks.

Disclaimers

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

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