Obsidian
AUDITS

Tenor Security Review

Auditors

0xjuaan

0xSpearmint

April 16, 2026

Introduction

Obsidian Audits

Obsidian audits is a team of top-ranked security researchers, with a publicly proven track record, specialising in DeFi protocols across EVM chains and Solana.

The team has achieved 10+ top-2 placements in audit competitions, placing 1st in competitions for Wormhole, Pump.fun, Yearn Finance, and many more.

Find out more: obsidianaudits.com

Audit Overview

Tenor offers a powerful interface for borrowers and lenders to access Morpho Midnight's fixed-rate, fixed-term markets. With Tenor, users can opt in to features like auto-renewal at maturity, conditional execution of orders, offers with bespoke terms, and more.

Tenor engaged Obsidian Audits to perform a security review of the smart contracts in the `tenor-morpho-v2-contracts-2` repo. The review was conducted from April 7, 2026 to April 16, 2026.

Scope

Files in scope

Repo	Files in scope
`tenor-morpho-v2-contracts-2` **Commit hash:** 1f75249414b61828c561e61f927fc0b365e41da8	`src/**/*.sol` (excluding `src/external`, `src/periphery/clamps`, `src/adapter/MidnightAdapterBase.sol`)

Summary of Findings

Severity Breakdown

A total of **10** issues were identified and categorized based on severity:

- **1 High severity**
- **5 Medium severity**
- **2 Low severity**
- **2 Informational**

Findings Overview

ID	Title	Severity	Status
H-01	Incorrect effective price formulas undercharge protocol fees	High	Fixed
M-01	`_validatePrice` does not account for Morpho V2's continous fee	Medium	Fixed
M-02	`_capTakeUnits` does not account for the fee charged in Tenor callbacks	Medium	Fixed
M-03	A borrower can enter a liquidation grace period even when their position is healthy	Medium	Fixed
M-04	Caller controlled `onBehalf` parameter in `midnightSupplyCollateral` enables collateral bitmap griefing	Medium	Fixed
M-05	Lenders can be renewed into obligations which will realize bad debt	Medium	Acknowledged

L-01	VaultV2 rounding mismatch leaves 1 wei of loan token stuck in callback	Low	Fixed
L-02	Borrower migration to lower-LTV market can leave position near liquidation	Low	Fixed
I-01	Priority liquidator inaction during grace period can lead to bad debt	Informational	Fixed
I-02	Inconsistent zero-amount fee check allows transfer revert	Informational	Fixed

Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High -** leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium -** leads to a moderate material loss of assets in the protocol or moderately harms a group of users. Alternatively, breaking a core aspect of the protocol's intended functionality
- **Low -** leads to a minor material loss of assets in the protocol or harms a small group of users.

Likelihood

- **High -** attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- **Medium -** the attack/vulnerability requires minimal or no preconditions, but there is limited or no incentive to exploit it in practice
- **Low -** requires highly unlikely precondition states, or requires a significant attacker capital with little or no incentive.

Findings

[H-01] Incorrect effective price formulas undercharge protocol fees

Description

The Tenor protocol charges a fee as a percentage of the interest on position renewals. Interest per unit is `(WAD - price)`, and the fee per unit is `feeRate * (WAD - price) / WAD`, denoted `x` in the code. The total fee should therefore be:

```
fee = obligationUnits * feePerUnit / WAD
fee = obligationUnits * feeRate * (WAD - price) / WAD / WAD
```

Instead of computing the fee directly, `CallbackLib` derives it through a "buyer effective price", the all-in price per unit the buyer pays including fees. The fee is then extracted as a residual:

```
fee = obligationUnits * (buyerEffectivePrice / WAD) - buyerAssets
```

Since the buyer pays `price` per unit for the bond plus `x` per unit in fees, the correct effective price is `price + x`. The current implementation instead computes `price * (WAD + 2x) / (WAD + x)`, which is not algebraically equivalent and produces a smaller fee for any `price`.

The same issue exists in `sellerEffectivePrice`, which computes `price * WAD / (WAD + x)` instead of the correct `price - x`.

Proof of Concept

Add the following logging to `test_onBuy_happyPath_withFee` in `test/unit/MorphoV2ToV2LendWithdrawable.t.sol`:

```
        // 2. Fee paid to recipient (exact amount)
        uint256 feeReceived =
loanToken.balanceOf(feeRecipient) - feeRecipientBefore;
+       uint256 totalInterest = result.obligationUnits -
result.buyerAssets;
+       console.log("obligationUnits:",
result.obligationUnits);
+       console.log("buyerAssets:    ",
result.buyerAssets);
+       console.log("total interest: ", totalInterest);
+       console.log("interest / 2:   ", totalInterest / 2);
+       console.log("fee received:   ", feeReceived);
        assertEq(feeReceived, expectedFee, "Fee recipient
should receive exact fee");
```

Console output:

```
Ran 1 test for
test/unit/MorphoV2ToV2LendWithdrawable.t.sol:MorphoV2ToV2Len
dWithdrawableTest
[PASS] test_onBuy_happyPath_withFee() (gas: 382267)
Logs:
  obligationUnits: 50493829654016279211
  buyerAssets:     50000000000000000000
  total interest:  493829654016279211
  interest / 2:    246914827008139605
  fee received:    243310213058145716
```

With a `feeRate` of 50%, the fee should be exactly half of the total interest. The fee received (243310213058145716) is ~1.46% less than the expected value (246914827008139605). The discrepancy grows significantly for lower prices.

Recommendation

Change `effPrice` in `buyerEffectivePrice` from `price.mulDivUp(WAD + 2 * x, WAD + x)` to `price + x`, and `effPrice` in `sellerEffectivePrice` from `price.mulDivDown(WAD, WAD + x)` to `price - x`.

Since x is the fee per unit, the all-in price is simply the base price plus (or minus for the seller) the fee component:

```
 function buyerEffectivePrice(uint256 price, uint256
feeRate) internal pure returns (uint256 effPrice) {
     if (feeRate == 0) return price;
     uint256 x = _interestFeeComponent(price, feeRate);
-    effPrice = price.mulDivUp(WAD + 2 * x, WAD + x);
+    effPrice = price + x;
 }
```

```
 function sellerEffectivePrice(uint256 price, uint256
feeRate) internal pure returns (uint256 effPrice) {
     if (feeRate == 0) return price;
     uint256 x = _interestFeeComponent(price, feeRate);
-    effPrice = price.mulDivDown(WAD, WAD + x);
+    effPrice = price - x;
 }
```

Additionally, consider updating `buyerFeeFromTick` and `sellerFeeFromTick` to directly calculate:

`fee = obligationUnits * feeRate * (WAD - price) / WAD / WAD` `fee = obligationUnits * x / WAD` , rather than deriving the fee through the effective price residual.

```
 function sellerFeeFromTick(uint256 tick, uint256 feeRate,
uint256 units, uint256 assets)
     internal
     pure
     returns (uint256)
 {
     if (feeRate == 0) return 0;
     uint256 price = TickLib.tickToPrice(tick);
-    uint256 effPrice = sellerEffectivePrice(price,
feeRate);
-    return sellerFee(units, assets, effPrice);
+    uint256 x = _interestFeeComponent(price, feeRate);
+    return units.mulDivUp(x, WAD);
 }

 function buyerFeeFromTick(uint256 tick, uint256 feeRate,
uint256 units, uint256 assets)
     internal
```

```
        pure
        returns (uint256)
    {
        if (feeRate == 0) return 0;
        uint256 price = TickLib.tickToPrice(tick);
-       uint256 effPrice = buyerEffectivePrice(price, feeRate);
-       return buyerFee(units, assets, effPrice);
+       uint256 x = _interestFeeComponent(price, feeRate);
+       return units.mulDivUp(x, WAD);
    }
```

Remediation: Fixed in commit ef231eb

> **Note:** the original finding was referring to the fact that the `feeRate` was not treated as the portion of interest paid (since it's treated as a portion of the APY instead), while the comments made it seem otherwise.
>
> Based on this message, we see that it's intended for the `feeRate` to represent the percentage diff in interest rate, rather than percentage diff in absolute interest.
>
> Though we do confirm that the `buyerEffectivePrice()` calculation was incorrect for a different reason, and has been fixed here.

[M-01] `_validatePrice` does not account for Morpho V2's continous fee

Description

`RenewalOrchestrator._validatePrice` enforces the user's `limitRatePerSecond`, their minimum acceptable yield, but does not account for Midnight's per-obligation `continuousFee`. A lender can accept a renewal offer that satisfies Tenor's gross-rate check, but yields less than their stated minimum once the continuous fee accrues.

Midnight stamps a fresh `pendingFee` on a lender's position every time they gain credit via `take()`:

```
uint128 buyerPendingFeeIncrease =
    UtilsLib.toUint128(buyerCreditIncrease.mulDivDown(
        _obligationState.continuousFee * timeToMaturity,
WAD));
buyerPos.pendingFee += buyerPendingFeeIncrease;
```

The lender's true face value at maturity is `credit − pendingFee`. The `continuousFee` is per-obligation, capped at `MAX_CONTINUOUS_FEE ≈ 1%`, and can be adjusted at any time by the `feeSetter`.

Tenor's rate validation only checks the gross bond yield against the user's limit. The `fee` parameter is Tenor's own callback fee, not Midnight's continuous fee.

Example scenario: A lender sets `limitRatePerSecond = 5% APR` on a `takeLendV2ToV2Withdrawable` callback. The target obligation has `continuousFee = MAX_CONTINUOUS_FEE = 1% APR`. A keeper presents an offer at exactly 5% APR gross. Tenor's `_validatePrice` passes. The lender accepts. At target maturity, the lender's net yield is ~4% APR — below their configured 5% minimum.

Recommendation

In `takeLendV2ToV2Withdrawable` and `takeLendV1ToV2`, including the obligation's `continuousFee` during validation and decrement `obligationUnits` by the expected continuous-fee accrual.

```
uint32 targetContinuousFee =
MORPHO_MIDNIGHT.continuousFee(targetObligationId);
if (targetContinuousFee > 0) {
    uint256 fullTimeToTargetMaturity = targetMaturity -
block.timestamp;
    uint256 continuousFeeAmount = obligationUnits.mulDivUp(
        uint256(targetContinuousFee) *
fullTimeToTargetMaturity, WAD
    );
    obligationUnits -= continuousFeeAmount;
}

_validatePrice(
    isBuy, obligationUnits, buyerAssets, fee,
    renewalParams.limitRatePerSecond, policyRate,
secondsToMaturity
);
```

This aligns `obligationUnits` with what the lender actually receives at maturity (face value = `credit - pendingFee`).

Remediation: Fixed in PR #362

[M-02] `_capTakeUnits` does not account for the fee charged in Tenor callbacks

Description

In the `TakeRouter`, the `_capTakeUnits` function is used to limit the obligation units of a `take()` on Midnight.

```
if (remaining < type(uint256).max / WAD) {
    takeUnits = UtilsLib.min(
        takeUnits, _remainingToUnits(taker, action.offer,
fillIndex, remaining, obligationId)
    );
}
```

The `takeUnits` is capped based on the `remaining` assets that can be filled (based on `maxFill`), but `_remainingToUnits` does not account for the Tenor fee. The orchestrator / `outputResolver` then adjusts the returned amounts by the fee *after* the cap, so the tracked `totals[fillIndex]` drifts away from the raw Midnight amount that was sized for.

Impact

The direction of the fee adjustment depends on the `fillIndex`:

`fillIndex`	**Fee effect on tracked total**	**Result**
`FILL_BUYER_ASSETS`	`buyerAssets += fee`	**Overshoot →** `FillOvershoot` **revert**
`FILL_SELLER_ASSETS`	`sellerAssets -= fee`	Silent underfill (batch completes below `maxFill`)
`FILL_OBLIGATION_UNITS`	no fee on units	no issue

There are two distinct impacts:

1. **DoS on multi-action batches** when the fee inflates the taker's tracked side, the last action pushes `totals[fillIndex]` past `maxFill` and the

batch reverts with `FillOvershoot`

2. **Silent underfill** when the fee deflates the tracked side — the batch fills strictly less than the caller requested.

Both `_dispatchOrchestrator` and `_dispatchMidnightTake` (via the `outputResolver`) are affected, since the cap happens before the fee adjustment in both paths.

Recommendation

Deflate (or inflate) `remaining` by the Tenor fee before passing it to `_remainingToUnits` when the fill dimension an asset rather than obligation units.

Remediation: Fixed in PR #379

[M-03] A borrower can enter a liquidation grace period even when their position is healthy

Description

`DelayedLiquidationGate.startGracePeriod()` requires that the borrower's position is healthy, using the following check:

```
if (MIDNIGHT.isHealthy(obligation, obligationId, borrower))
{
    revert PositionIsHealthy();
}
```

The issue is that a borrower can make their position temporarily unhealthy (by calling `take()` to borrow loan tokens without supplying enough collateral).

During this transient unhealthy state, they can call `DelayedLiquidationGate.startGracePeriod()` during the `onSell` callback (which will succeed since the position is temporarily unhealthy), before repaying the under-collateralised loan, to ensure the health check at the end of `take()` succeeds.

This breaks the invariant `GATE-3`, which states that only unhealthy positions can enter a grace period.

Recommendation

One (restrictive) solution would be to enforce that the caller of `startGracePeriod()` is an EOA (without EIP-7702 delegations):

```
require(tx.origin == msg.sender && msg.sender.code.length ==
0
```

Remediation: Fixed in PR #372

[M-04] Caller controlled `onBehalf` parameter in `midnightSupplyCollateral` enables collateral bitmap griefing

Description

Morpho Midnight limits each borrower to 10 active collateral types per obligation (`MAX_COLLATERALS_PER_BORROWER`). To prevent griefing, `supplyCollateral` requires authorization: `require(onBehalf == msg.sender || isAuthorized[onBehalf][msg.sender])`. This stops anyone from activating unwanted bitmap slots on a victim's position.

`MidnightAdapterBase.midnightSupplyCollateral` accepts `onBehalf` as a parameter rather than pinning it to `initiator()`. Every user who wants to use the bundler for withdrawals or collateral operations must authorize the TenorAdapter on Midnight.

As a result an attacker can exploit this by calling `Bundler3.multicall` with a bundle that transfers 1 wei of a collateral token to the adapter and then calls `midnightSupplyCollateral(obligation, collateralIndex, 1, victim)`. Midnight sees `msg.sender = TenorAdapter`, which the victim authorized, and accepts the supply. Repeating across collateral indices fills all 10 bitmap slots with dust, blocking the victim from adding the collateral they actually need.

This is a griefing attack with no direct fund loss. The victim can withdraw the dust collateral to free collateral slots.

Recommendation

Pin `onBehalf` to `initiator()` in `midnightSupplyCollateral`, matching how `midnightWithdrawCollateral` and `midnightWithdraw` already handle it. Remove the `onBehalf` function parameter.

```
    function midnightSupplyCollateral(
        Obligation calldata obligation,
        uint256 collateralIndex,
        uint256 assets,
-       address onBehalf
    ) external onlyBundler3 {
-       require(onBehalf != address(0), ErrorsLib.ZeroAddress());
-       require(onBehalf != address(this), ErrorsLib.AdapterAddress());
+       address onBehalf = initiator();
```

Remediation: Fixed in PR #382

[M-05] Lenders can be renewed into obligations which will realize bad debt

Description

Any lender who has configured `RenewalParams` with a `targetMarketId` whose obligation includes a particular collateral token (e.g. USR) can be forced into that market by an attacker after the collateral token loses its value.

The attacker exploits the auto-renewal mechanism to move the lender's funds into an obligation backed by worthless collateral, causing the lender to lose their principal.

Using an `OracleWithValidationCheck` (which can be used to revert when oracle and DEX prices diverge) does **not** fully mitigate this, because the oracle price is only queried when `debt > 0`. A lender-to-lender renewal (`takeLendV2ToV2Withdrawable`) where the counterparty (seller) already holds credit bypasses the oracle check entirely. In a lender-to-lender transfer, the counterparty's `_position.credit` decreases but their `_position.debt` does not increase, so the price query in `isHealthy` is skipped and the `OracleWithValidationCheck` never gets a chance to revert.

Example scenario:

1. An obligation exists with `loanToken = USDC` and `collateral = USR`. A lender authorizes renewals into this `targetMarketId` with `takeGate = address(0)`.
2. USR depegs, and its collateral is now worthless.
3. An attacker who already has lending credit on the USR-collateralised obligation signs a **sell offer** to offload that credit.
4. The attacker calls `takeLendV2ToV2Withdrawable`. The lender's USDC is withdrawn from their source obligation and sent to the attacker, while the lender receives credit backed by worthless USR collateral.

Recommendation

Consider creating a `TakeGate` implementation that validates the collateral quality of the target obligation at renewal time. The gate should:

1. Compare oracle price to a DEX/spot price for each collateral token in the target obligation (by querying the oracle price)
2. Verify the target obligation's oracle is a trusted `OracleWithValidationCheck`, rejecting renewals into obligations using oracles that lack depeg protection

Using such a `TakeGate` in the user's renewal params will prevent this issue.

Remediation: Issue acknowledged

[L-01] VaultV2 rounding mismatch leaves 1 wei of loan token stuck in callback

Description

The (CB-DUST-1) property states: "After callback completes, the callback contract holds no token balances."

However `MorphoV2WithdrawVaultSharesCallback` violates this when used with a VaultV2 vault.

The callback computes shares via `sharesToWithdraw = previewWithdraw(buyerAssets)`, then calls `redeem(sharesToWithdraw)`.

VaultV2's `previewWithdraw` uses:

```
assets.mulDivUp(newTotalSupply + virtualShares,
newTotalAssets + 1)
```

and `redeem` internally calls `previewRedeem` which computes assets returned via:

```
shares.mulDivDown(newTotalAssets + 1, newTotalSupply +
virtualShares)
```

`previewWithdraw` rounds up to guarantee the shares redeem to at least `buyerAssets`. This means `sharesToWithdraw` can be a share count that actually redeems for more than `buyerAssets`. The callback only approves exactly `buyerAssets` for Midnight to pull, so the excess (1 wei) remains stuck in the callback.

Recommendation

Replace `redeem(sharesToWithdraw)` with `withdraw(buyerAssets)` so the vault transfers exactly `buyerAssets` to the callback and no dust remains.

Remediation: Fixed in PR #383

[L-02] Borrower migration to lower-LTV market can leave position near liquidation

Description

When a borrower renewal executes across any migration path (V1-to-V2, V2-to-V1, V2-to-V2), collateral is migrated pro-rata from the source to the target. If the target market has a lower LLTV than the source, the borrower's position can land just above the liquidation threshold on the target, making them liquidatable shortly after from minor price movement or interest accrual.

The borrower called `setUserRenewalParams` for this source/target pair, so they are aware of the target market's LLTV. However, a borrower approving a renewal offer likely does not intend for the migration itself to place them at the edge of liquidation.

Recommendation

Document this risk clearly for borrowers setting up renewal offers across markets with different LLTVs. The borrower can configure a `takeGate` in their `UserRenewalParams`, but no health-factor gate exists today. Consider implementing one that reverts the migration if the resulting target position LTV exceeds a borrower-specified threshold.

Remediation: Fixed in PR #384

[I-01] Priority liquidator inaction during grace period can lead to bad debt

Description

The `DelayedLiquidationGate` gives a priority liquidator exclusive access for `PRIORITY_PERIOD` after the grace period ends. During this window, no other liquidator can act. If the priority liquidator fails to liquidate (offline, or malicious), the position remains unliquidated for the combined duration of `GRACE_PERIOD + PRIORITY_PERIOD` before other liquidators can step in.

With volatile collateral, this delay can be enough for the collateral value to drop below the outstanding debt, resulting in bad debt.

Recommendation

Consider allowing any liquidator to act immediately after the grace period ends, removing exclusive priority access.

If priority exclusivity is necessary, consider clearly documenting the bad debt risk and conveying it to lenders.

Remediation: Fixed in PR #378

[I-02] Inconsistent zero-amount fee check allows transfer revert

Description

`MorphoV2ToV1LendCallback.onSell` calculates a flat percentage fee on `sellerAssets` using `CallbackLib.percentageFee`, which rounds down via `mulDivDown`. The transfer is guarded by `if (callbackData.feeRate > 0)`, which checks whether a fee rate is configured, not whether the computed fee is non-zero. If `sellerAssets * feeRate < 1e18`, the fee rounds to zero and `safeTransfer` is called with amount zero. Some ERC20 tokens (e.g. BNB) revert on zero-amount transfers, preventing the lender's V2-to-V1 migration from occuring.

All five other callbacks in the codebase guard the transfer with `if (fee > 0)`, checking the computed value.

Recommendation

Compute `fee` unconditionally and wrap the `safeTransfer` in `onSell` with `if (fee > 0)`, matching the pattern used in the other callbacks.

```
  uint256 fee;
  if (callbackData.feeRate > 0) {
      fee = CallbackLib.percentageFee(sellerAssets,
callbackData.feeRate);
-     SafeTransferLib.safeTransfer(obligation.loanToken,
callbackData.feeRecipient, fee);
+ }
+ if (fee > 0) {
+     SafeTransferLib.safeTransfer(obligation.loanToken,
callbackData.feeRecipient, fee);
  }
```

Remediation: Fixed in commit 05c4e4b