Blackthorn

Security Review For **Morpho**

Collaborative Audit Prepared For:	**Morpho**
Lead Security Expert(s):	**0x73696d616f**
	hyh
	pkqs90
	xiaoming90
Date Audited:	**April 6 - April 22, 2026**

Introduction

Morpho Midnight is a non-custodial fixed-rate lending protocol implemented for the Ethereum Virtual Machine. It is organized around isolated, immutable, permissionlessly created markets with fixed-maturity. Lending and borrowing are implemented through the trading of credit and debt units, whose payoff structure is analogous to that of zero-coupon obligations, settling at the market's maturity. Participants trade by posting or consuming offers that do not lock capital and source liquidity only at settlement, allowing makers to quote across multiple markets at once. Markets can range from single to multi-collateral configurations, and gates can be used to implement access-control policies.

Scope

Repository: morpho-org/midnight

Audited Commit: 8de7076fb1edd52c39694ba09c65f6bbef2a9c25

Final Commit: e6f2bf28400d8215a533f713081a6f68ca121941

Files:

- src/authorizers/EcrecoverAuthorizer.sol
- src/interfaces/ICallbacks.sol
- src/interfaces/IEcrecover.sol
- src/interfaces/IERC20.sol
- src/interfaces/IGate.sol
- src/interfaces/IMidnight.sol
- src/interfaces/IOracle.sol
- src/interfaces/IRatifier.sol
- src/libraries/ConstantsLib.sol
- src/libraries/EventsLib.sol

- src/libraries/IdLib.sol
- src/libraries/SafeTransferLib.sol
- src/libraries/TickLib.sol
- src/libraries/UtilsLib.sol
- src/Midnight.sol
- src/periphery/TakeAmountsLib.sol
- src/ratifiers/ApprovalRatifier.sol
- src/ratifiers/EcrecoverRatifier.sol

Final Commit Hash

e6f2bf28400d8215a533f713081a6f68ca121941

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.

Medium Severity Findings: 2

Low/Informational Severity Findings: 25

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Security Experts Dedicated to This Review

@0x73696d616f

@hyh

@pkqs90

@xiaoming90

Issue M-1: In `lossIndex == type(uint128).max` state a fresh lender principal can be slashed to zero on entry [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/23

Summary

When bad debt equals all outstanding units, `lossIndex` saturates to `type(uint128).max` and `totalUnits` drops to zero. The obligation is economically terminal but `take()` still allows entry. A late lender's fresh credit is zeroed by the next `_updatePosition` because the saturated obligation level loss index is applied to the new position.

Vulnerability Detail

`Midnight.sol#L547-L556`: bad-debt branch updates `lossIndex` to `type(uint128).max` when `badDebt == oldTotalUnits`. `Midnight.sol#L321-L345`: `take()` doesn't check for terminal `lossIndex` state before allowing credit/debt entry.

When a prior liquidation realizes `badDebt == oldTotalUnits`, saturating `lossIndex`, the same obligation id remains usable in later `take()` calls, and there is no `ObligationTerminated` kind of event emitted at the saturating write.

Collateral price can recover after saturation, so the market can appear usable. E.g., a victim lender then can sign a buy offer on the terminal obligation, and this investment won't be recoverable once order becomes filled.

Impact

Credit of newly entered lenders will be slashed fully, as if they had lived through the old loss event, even though they entered afterwards. I.e. full lender principal is lost per victim entry in any terminal obligation.

Code Snippet

`lossIndex` update formula in `liquidate()` runs when `badDebt > 0`:

Midnight.sol#L552-L556

```
_obligationState.lossIndex = UtilsLib.toUint128(
    type(uint128).max - (type(uint128).max - oldLossIndex).mulDivDown(oldTotalUnits
    ↪  - badDebt, oldTotalUnits)
);
_obligationState.totalUnits -= UtilsLib.toUint128(badDebt);
```

If `badDebt == oldTotalUnits`, the multiplicative term becomes zero and `lossIndex` is set to `type(uint128).max`, and after that this term stays zero.

Tool Used

Manual Review

Recommendation

Consider rejecting a new credit or debt entry when `lossIndex` is saturated, e.g.:

```
    uint256 buyerCreditIncrease = UtilsLib.zeroFloorSub(units, buyerPos.debt);
    uint256 sellerCreditDecrease = UtilsLib.min(units, sellerPos.credit);
    uint256 sellerDebtIncrease = units - sellerCreditDecrease;
+   require(
+       _obligationState.lossIndex != type(uint128).max || (buyerCreditIncrease ==
 ↪  0 && sellerDebtIncrease == 0),
+       "terminal obligation"
+   );
```

Consider emitting `ObligationTerminated(id)` in the saturating liquidation branch.

Discussion

MathisGD

fixed by https://github.com/morpho-org/midnight/pull/743

Issue M-2: `take()` does not control near-maturity fills, allowing unconsented seller debt and post-maturity collateral seizure [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/40

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

Summary

If an offer remains fillable until very close to `offer.obligation.maturity`, a taker can attack the seller side of that fill by calling `Midnight.take()` directly, not through the router, at the last allowed timestamp and then liquidating the newly-created position as overdue some blocks later.

The core issue is that `take()` only enforces `block.timestamp <= offer.expiry`, it doesn't enforce any minimum time-to-maturity and does not forbid near-maturity fills from creating new seller debt. This is especially dangerous when `offer.expiry == offer.obligation.maturity`: the fill is still accepted at exact maturity, because `take()` uses `<= offer.expiry`, while `isLiquidatable()` only treats a position as overdue once `block.timestamp > obligation.maturity`.

As a result, when the maker is the actual seller, a taker can fill the maker's near-maturity order at the boundary, wait until maturity has passed, and liquidate the resulting position for the post-maturity LIF bonus. The same root cause also covers the continuous-fee shortfall path at maturity.

Vulnerability Detail

`take()` validates the offer against `start` and `expiry`, but it never validates that the fill is still safely far enough from `offer.obligation.maturity`:

- if `offer.expiry` is equal to or just before maturity, the order is still fillable;
- if the fill creates `sellerDebtIncrease > 0`, the seller becomes a borrower;
- if the fill happens at exact maturity, it still passes the end-of-function solvency check because overdue liquidation only starts once `block.timestamp > obligation.maturity`;
- one or a few blocks later, the position is overdue and can be liquidated for the post-maturity LIF bonus.

This is not prevented by the router because the attacker can call the core `take()` entrypoint directly.

In the generic path, the issue is:

1. `take()` accepts the fill because `block.timestamp <= offer.expiry`.
2. `_updatePosition()` runs before settlement deltas are finalized.
3. `sellerDebtIncrease = units - sellerCreditDecrease` is computed and recorded without a dedicated near-maturity guard.
4. `take()` only checks that the seller is not liquidatable at the time of the fill.
5. At exact maturity, that check still passes because `isLiquidatable()` uses strict `>`.
6. Once time advances past maturity, the seller's new debt is overdue and can be liquidated.

There is also continuous fee related mechanics for at maturity `take()`:

1. Maker has credit and collateral in a continuous-fee obligation.
2. Market maker signs a full-credit sell offer with `maxUnits = credit` and default `reduceOnly = false`.
3. Taker fills at exact maturity.
4. `_updatePosition()` accrues the remaining fee, reducing credit to `credit - fee`.
5. `take()` consumes all updated credit and records the accrued `pendingFee` shortfall as `sellerDebtIncrease`.

6. After maturity, in a few blocks, the unintended debt is liquidated for LIF bonus against the maker's collateral.

It is the same near-maturity fill issue, but with continuous-fee accrual providing the shortfall that turns a credit exit into debt.

The `expiry == maturity` case deserves explicit documentation being both natural and dangerous, as it is intuitive for users and integrators to keep an offer valid "until maturity", but in this implementation that means the offer is still fillable at exact maturity and exact maturity is precisely the boundary where the position can be created, pass `isLiquidatable()`, and then become overdue in the next block.

Impact

This can cause direct collateral loss to the seller whose near-maturity fill creates debt.

E.g., for the fresh-debt path, at `LLTV = 0.77` and `LIQUIDATION_CURSOR_LOW = 0.25`, post-maturity `maxLif` is about `1.061006`, so the liquidator captures about a `6.1%` bonus over repaid debt.

Code Snippet

Midnight.sol#L265-L266

```
require(block.timestamp >= offer.start, "offer not started");
require(block.timestamp <= offer.expiry, "offer expired");
```

Midnight.sol#L324-L342

```
if (hasCredit(id, buyer) || units > buyerPos.debt)
↪  _updatePosition(offer.obligation, id, buyer);
if (hasCredit(id, seller)) _updatePosition(offer.obligation, id, seller);

uint256 buyerCreditIncrease = UtilsLib.zeroFloorSub(units, buyerPos.debt);
uint256 sellerCreditDecrease = UtilsLib.min(units, sellerPos.credit);
uint256 sellerDebtIncrease = units - sellerCreditDecrease;

sellerPos.pendingFee -= sellerPendingFeeDecrease;
```

```
sellerPos.credit -= UtilsLib.toUint128(sellerCreditDecrease);
sellerPos.debt += UtilsLib.toUint128(sellerDebtIncrease);
```

Midnight.sol#L403-L403

```
require(!isLiquidatable(offer.obligation, id, seller), "seller is liquidatable");
```

Midnight.sol#L835-L837

```
function isLiquidatable(Obligation memory obligation, bytes32 id, address borrower)
↪  public view returns (bool) {
    return position[id][borrower].debt > 0 && !liquidationLocked(id, borrower)
        && (block.timestamp > obligation.maturity || !isHealthy(obligation, id,
        ↪  borrower));
}
```

Tool Used

Manual Review

Recommendation

Consider documenting these risks and adding a minimal distance between `block.timestamp` and `offer.obligation.maturity`, not allowing exposure-increasing fills to happen at or too close to maturity.

Discussion

MathisGD

we decided to not document as we don't see the behavior as being misleading in any way.

Issue L-1: Chain fork changes all obligation Ids, making them unreachable [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/19

Summary

`IdLib.toId()` includes `block.chainid` in the obligation id computation, so after a chain fork, all existing obligations become unreachable and interactions silently create new obligations with fresh state.

Vulnerability Detail

The obligation id is computed here as:

```
function toId(Obligation memory obligation, uint256 chainId, address midnight)
 ↪  internal pure returns (bytes32) {
    return keccak256(
        abi.encodePacked(
            uint8(0xff), midnight, chainId,
             ↪  keccak256(abi.encodePacked(SSTORE2_PREFIX, abi.encode(obligation)))
        )
    );
}
```

After a chain fork, `block.chainid` changes, producing a different id for the same obligation parameters. All storage (positions, collateral, credit, debt, withdrawable) is keyed by the old id, but every function recomputes the id via `toId()`, so the old state is unreachable.

When users interact with the same obligation post-fork, `touchObligation()` creates a brand new obligation with zero state. This does not revert, silently splitting the protocol into an orphaned old obligation (with all the funds) and a fresh new one (with no funds).

This is especially problematic for liquidations, as when it comes back up after a code fix,

some positions may be instantly liquidatable due to accrued interest or oracle price changes during the downtime.

Impact

All funds (loan tokens, collateral) under existing obligations are permanently locked. Lenders lose their credit, borrowers lose their collateral. Very unlikely to happen, and if it does, it should be announced beforehand so the code can be fixed.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/main/morpho-org__midnight/src/libraries/IdLib.sol#L23-L29

Tool Used

Manual Review

Recommendation

Remove `chainId` from the id computation, similar to Morpho Blue which does not include it in the market id. Cross-chain replay protection is already handled by the EIP-712 domain separator in the ratifier and authorizer contracts.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/756

Issue L-2: `testReturnJumps` asserts the wrong ratio direction [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/20

Summary

`testReturnJumps` asserts `currentReturn / previousReturn ~= 1.025`, but the actual ratio is `~0.975` since higher tick means higher price and lower return. The test only passes due to the 10% tolerance.

Vulnerability Detail

In TickLibTest.sol#L33:

```
function testReturnJumps() public pure {
    for (uint256 i = 220; i <= 770; i++) {
        uint256 previousReturn = _return(TickLib.tickToPrice(i - 1));
        uint256 currentReturn = _return(TickLib.tickToPrice(i));
        assertApproxEqRel(
            currentReturn.mulDivDown(1e18, previousReturn), 1.025e18, 0.1e18, ...
        );
    }
}
```

`_return(price) = 1e18 / price - 1`, which is `1/price - 1`. Since `tickToPrice` is monotonically increasing, a higher tick gives a higher price and a lower return. Therefore `currentReturn / previousReturn = 1/1.025 ~= 0.9756`, not `1.025`.

The assertion passes because `|0.9756 - 1.025| / 1.025 ~= 4.8%`, which is within the 10% relative tolerance.

Impact

The test does not verify what it intends to. The expected value should be `~0.9756e18` (i.e. `1/1.025`), or the ratio should be flipped to `previousReturn / currentReturn`.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/main/morpho-org__midnight/test/TickLibTest.sol#L28-L36

Tool Used

Manual Review

Recommendation

Fix the assertion to check the correct ratio:

```
assertApproxEqRel(
    previousReturn.mulDivDown(1e18, currentReturn), 1.025e18, 0.01e18, ...
);
```

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/683

Issue L-3: Midnight constructor can use `msg.sender` instead of reading `roleSetter` from storage [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/21

Summary

The constructor emits `roleSetter` (an SLOAD) in the event instead of `msg.sender`, wasting gas since they are the same value.

Vulnerability Detail

In the constructor:

```
constructor() {
    roleSetter = msg.sender;
    emit EventsLib.Constructor(roleSetter);
}
```

`roleSetter` was just assigned `msg.sender`, so the emit reads back from storage instead of using the already-available `msg.sender` from the calldata.

Impact

Wastes gas on an unnecessary SLOAD in the constructor.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/main/morpho-org__midnight/src/Midnight.sol#L133-L136

Tool Used

Manual Review

Recommendation

```
constructor() {
    roleSetter = msg.sender;
    emit EventsLib.Constructor(msg.sender);
}
```

Discussion

MathisGD

we don't really optimize for constructor gas efficiency. Though it's better to not use immutable variables in the constructor in general, as it's not clear if they have already been set of not

fixed in https://github.com/morpho-org/midnight/pull/748

Issue L-4: `LLTV_8 = WAD` eliminates post-maturit liquidation incentive [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/22

Summary

The allowed LLTV tier `LLTV_8 = 1e18` causes `maxLif(WAD, cursor) == WAD` for all cursor values. Post-maturity liquidators receive only par collateral for par debt, making liquidation negative EV after gas. Lenders will have to liquidate themselves as they are the only incentivized actors left.

Vulnerability Detail

Root Cause

`/src/libraries/ConstantsLib.sol:17-31`: `LLTV_8 = 1e18` is in the allowed set. The LIF formula `WAD.mulDivDown(WAD, WAD - cursor.mulDivDown(WAD - lltv, WAD))` collapses to `WAD` when `lltv == WAD`, producing zero liquidation bonus at `/src/Midnight.sol:860-862`.

Internal Pre-conditions

1. Obligation uses a collateral parameter with `lltv == LLTV_8`
2. `maxLif` for both `LIQUIDATION_CURSOR_LOW` and `LIQUIDATION_CURSOR_HIGH` equals `WAD`
3. Borrower has live debt in that obligation

External Pre-conditions

1. Borrower does not voluntarily repay at maturity (rational if collateral lost value)
2. Rational liquidators require positive spread to cover gas/slippage

Attack Path

1. Borrower opens debt in an obligation whose collateral LLTV is `LLTV_8`
2. Correlated-peg collateral (e.g. stETH/ETH) depegs 2% at maturity
3. Borrower rationally abandons: repaying 10M ETH to recover 9.8M ETH of stETH is -200K loss; walking away is free
4. Post-maturity LIF ramps from `WAD` to `WAD` (zero bonus). `Midnight.sol:565-571`: `WAD + (WAD - WAD) * elapsed / TIME_TO_MAX_LIF == WAD`
5. Liquidation is negative EV after gas: `0` ETH liquidation bonus minus `0.006 ETH` estimated gas for a 200k-gas close at 30 gwei
6. Lender capital: 2% bad debt socialized immediately, remaining 98% frozen

Impact

Lenders in LLTV_8 obligations lose the protocol's post-maturity enforcement mechanism. E.g. for a $10M ETH/stETH position with 2% depeg: $200K bad debt socialized instantly, $9.8M frozen with no profitable close path. The liquidator EV is `-0.006 ETH` before slippage for the final close, so profit-seeking keepers rationally skip it.

Code Snippet

Incentive is a function of `lltv` and `cursor`, which collapses to `WAD` when LLTV is `LLTV_8`:

Midnight.sol#L859-L862

```
/// @dev Returns the max LIF for the given lltv and cursor.
function maxLif(uint256 lltv, uint256 cursor) public pure returns (uint256) {
    return WAD.mulDivDown(WAD, WAD - cursor.mulDivDown(WAD - lltv, WAD));
}
```

Tool Used

Manual Review

Recommendation

The straightforward take is to remove `LLTV_8` from the allowed set, e.g.:

```
 function isLltvAllowed(uint256 lltv) pure returns (bool) {
     return lltv == LLTV_0 || lltv == LLTV_1 || lltv == LLTV_2 || lltv == LLTV_3 ||
      ↪  lltv == LLTV_4 || lltv == LLTV_5
-        || lltv == LLTV_6 || lltv == LLTV_7 || lltv == LLTV_8;
+        || lltv == LLTV_6 || lltv == LLTV_7;
 }
```

Alternatively, consider documenting the behavior and advising the additional simulation analysis before using the parameter.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/749

Issue L-5: Grouped offers can exceed a maker's intended shared liquidity cap [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/24

Vulnerability Detail

`group` is meant to provide shared replay protection and enforce liquidity limits across multiple offers.

In the code, all same `group` offers are tracked through a single counter, `consumed[maker][group]`.

The issue is that `take()` does not keep this counter in the same semantics. Depending on the offer, it increments the same `consumed` value with `sellerAssets`, `buyerAssets`, or `units`:

```
if (offer.maxSellerAssets > 0) {
    newConsumed = consumed[offer.maker][offer.group] += sellerAssets;
    require(newConsumed <= offer.maxSellerAssets, "consumed seller assets");
} else if (offer.maxBuyerAssets > 0) {
    newConsumed = consumed[offer.maker][offer.group] += buyerAssets;
    require(newConsumed <= offer.maxBuyerAssets, "consumed buyer assets");
} else {
    newConsumed = consumed[offer.maker][offer.group] += units;
    require(newConsumed <= offer.maxUnits, "consumed units");
}
```

As a result, same `group` offers can compare incompatible quantities against the same accumulator.

Example: Alice wants to place two `buy` offers that share one 1,000 USDC budget, so she signs both with the same `group`. Here, Alice is the buyer/maker. Bob is the taker filling her offers from the other side. Assume the trade charges a 0.5% fee, so a fill can have `buyerAssets != sellerAssets`.

Offer A uses `maxSellerAssets = 995 USDC`, while Offer B uses `maxBuyerAssets = 1,000 US`

DC. Bob first fills Offer A for a trade where Alice pays buyerAssets = 1,000 USDC, but Bob only receives sellerAssets = 995 USDC, the 5 USDC difference being the fee. Because Offer A is capped by maxSellerAssets, take() adds only 995 to consumed[][], even though Alice already spent 1,000 USDC. Bob can then still fill another 5 USDC on Offer B before the shared counter reaches 1,000. Alice has now spent 1,005 USDC across the group despite intending a shared 1,000 USDC cap.

Impact

When incompatible same-group offers are signed, takers can overfill the maker's aggregate intended budget.

Recommendation

Bind each group to a single semantic.

Discussion

MathisGD

there is this in the docs:

```
/// @dev Groups are useful to have a global offered amount shared across multiple
 ↪  offers ("OCO").
/// @dev To work as expected, all offers in the same group should have the same max
 ↪  values and loan token.
```

don't you think that's enough?

we considered at some point having 3 groups all applying in parallel, but that expensive in gas...

Issue L-6: Mismatched obligation ids in `TakeAmountsLib` can cause incorrect calculation [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/25

Vulnerability Detail

`TakeAmountsLib` is meant to preview how many `units` are needed for a target amount of buyer or seller assets. However, `buyerAssetsToUnits()` and `sellerAssetsToUnits()` take both an `offer` and a caller-supplied `id`, then use `offer.obligation.maturity` together with `midnight.tradingFee(id, ...)` without checking that the `id` actually belongs to `offer.obligation`:

```
function buyerAssetsToUnits(Midnight midnight, bytes32 id, Offer memory offer,
↪  uint256 targetBuyerAssets) {
  ...
  uint256 tradingFee = midnight.tradingFee(id,
  ↪  UtilsLib.zeroFloorSub(offer.obligation.maturity, block.timestamp));
  ...
}
```

This creates an unsafe interface: the helper can quote `units` using one obligation's fee schedule while the real `take()` call later settles against another. The current consumer, `TakeBundler`, derives a single id from `takes[0].offer.obligation` and reuses it for later offers while only documenting that all offers should share the same obligation id, without enforcing it:

```
bytes32 id = midnight.touchObligation(takes[0].offer.obligation);
...
TakeAmountsLib.sellerAssetsToUnits(midnight, id, takes[i].offer, ...)
```

Even though there is a code comment saying that all offers should use the same

obligation, it is still not a recommended code practice.

Impact

`TakeAmountsLib` library may return incorrect calculation result if caller side isn't careful enough.

Recommendation

Remove the redundant `id` parameter from `TakeAmountsLib` and derive it from `offer.obligation` internally.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/755

we document because it saves gas, and there are other instances of this in Midnight

Issue L-7: Inconsistent behavior between `TakeAmountLib` helper and core settlement [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/26

Vulnerability Detail

In some states, it is possible that `offerPrice < tradingFee` on a buy offer in the `TakeAmountLib.buyerAssetsToUnits()` function.

The core settlement (`take()`) will explicitly revert due to underflow when perform the checked math operation. However, the `TakeAmountLib` helper does not. So, the helper can quote a fill for an order that cannot be executed in reality.

Assuming the max-fee, the following tick-range will see this issue:

- 0d / 1d fee 1.4e13: tick 0 - tick 65
- 7d fee 9.8e13: tick 0 - tick 148
- 30d fee 4.17e14: tick 0 - tick 207
- 90d fee 1.25e15: tick 0 - tick 252
- 180d fee 2.5e15: tick 0 - tick 280
- 360d+ fee 5e15: tick 0 - tick 308

For example, for 360d, `maxTradingFee(6) = 5e15`. Then `tickToPrice(308) = 4.925e15`, so `offerPrice < tradingFee`.

Thus, if the intention is to replicate the `take()` behavior, consider reverting if `offerPrice < tradingFee`.

Impact

The `TakeAmountLib` helper can quote a fill for an order that cannot be executed in reality, leading to unexpected revert.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/db47e3dc348d793337fc8c03d63f61ead0f3741d/TakeAmountsLib.sol#L16

Tool Used

Manual Review

Recommendation

Consider mirroring the math in the `take()` - indirect revert due to checked math:

```
uint256 sellerPrice = offer.buy ? offerPrice - tradingFee : offerPrice;
uint256 buyerPrice = sellerPrice + tradingFee;
require(buyerPrice <= WAD, "buyerPrice");
```

Or add a `require(tradingFee <= offerPrice)`.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/754

Issue L-8: Values returned by the helpers might unexpectedly trigger the gates [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/27

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

Vulnerability Detail

```
File: TakeAmountsLib.sol
28:     // Forward: sellerAssets = offer.buy ? units.mulDivDown(sellerPrice, WAD) :
↪  units.mulDivUp(sellerPrice, WAD).
29:     /// @dev Returns the number of units to take to get the target seller
↪  assets.
30:     function sellerAssetsToUnits(Midnight midnight, bytes32 id, Offer memory
↪  offer, uint256 targetSellerAssets)
31:         internal
32:         view
33:         returns (uint256)
34:     {
35:         uint256 offerPrice = TickLib.tickToPrice(offer.tick);
36:         uint256 tradingFee = midnight.tradingFee(id,
↪  UtilsLib.zeroFloorSub(offer.obligation.maturity, block.timestamp));
37:         uint256 sellerPrice = offer.buy ? offerPrice - tradingFee : offerPrice;
38:         return
39:             offer.buy ? targetSellerAssets.mulDivUp(WAD, sellerPrice) :
↪  targetSellerAssets.mulDivDown(WAD, sellerPrice);
40:     }
```

Assume `offer.buy == false`:

```
Core forward (ceiling-rounded) in take():
  sellerAssets = ceil(units * sellerPrice / WAD)
```

```
  buyerAssets  = ceil(units * buyerPrice  / WAD)

Inverse (floor-rounded) in TakeAmountsLib periphery:
  sellerAssetsToUnits(A) = floor(A * WAD / sellerPrice)
  buyerAssetsToUnits(A)  = floor(A * WAD / buyerPrice)
```

For some `A` (when `p < WAD`), multiple `units` values can map to the same `A`. Let that set of values be `plateau()`.

```
plateau(A) = [minimum, maximum]
plateau(A) = [floor((A-1)*WAD/p) + 1 , floor(A*WAD/p)]
```

The periphery helper picks the maximum. The difference `max - min` can be >= 1 units. Depend on the offer's configuraiton, there can be multiple differents values all map back to the same A.

Assume a lender with credit on the market, and he wants to exit and receive exactly 4.200002 USDC. This lender is gated and prevented from increase debt, so `canIncreaseDebt()` is false.

```
offer.buy        = false
sellerPrice      = 0.4e18 = 400_000_000_000_000_000
fund.credit      = 10_500_003 units
target assets    = 4_200_002 USDC
```

In this case, there are three values of units that actually map to 4_200_002 in the `take()`. Plateau = `{10_500_003, 10_500_004, 10_500_005}` = width 3.

units	`units * 0.4`	ceil = `sellerAssets`
10_500_002	4_200_000.8	4_200_001
10_500_003	4_200_001.2	4_200_002 (min)
10_500_004	4_200_001.6	4_200_002
10_500_005	4_200_002.0	4_200_002 (max)

units	`units * 0.4`	ceil = `sellerAssets`
10_500_006	4_200_002.4	4_200_003

And the `sellerAssetsToUnits()` helper function return the maximum value out of the three.

```
sellerAssetsToUnits(4_200_002)
  = floor(4_200_002 * 1e18 / 0.4e18)
  = floor(4_200_002 / 0.4)
  = floor(10_500_005)
  = 10_500_005 (max of plateau)
```

Some routers, bundler or takers discover this offer and decided to fill the entire 4_200_002 USDC via seller-assets-denominated route, so it called `sellerAssetsToUnits(4_200_002)` and get back `10_500_005` unit. One example can be found in `TakeBundler.bundleTakeSellerAssets()` function.

They will proceed to call `take(10_500_005)`, so:

```
sellerCreditDecrease = min(10_500_005, 10_500_003) = 10_500_003
sellerDebtIncrease = 10_500_005 - 10_500_003 = 2
```

Since `sellerDebtIncrease > 0`, `canIncreaseDebt()` will revert the transaction, which blocked the exit.

However, if the `sellerAssetsToUnits()` returns the minimum (10_500_003) instead of the maximum of the plateau, the exit via `take(10_500_003)` will execute without any issue.

The same issue applies to the `buyerAssetsToUnits()` function.

Impact

Any external party (e.g., router, bundler, vault, users) that depends on the `sellerAssetsToUnits` and `buyerAssetsToUnits` helpers might encountered unexpected revert in an edge case.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/periphery/TakeAmountsLib.sol#L25

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/periphery/TakeAmountsLib.sol#L39

Tool Used

Manual Review

Recommendation

The `sellerAssetsToUnits()` and `buyerAssetsToUnits()` should compute the smallest `units` value that still maps back to the requested asset amount under the core forward calculation

Discussion

MathisGD

> might unexpectedly trigger the gates

note that gating cannot depend on amounts, so not really.

it would be elegant to have the smallest units for sell and highest units for buy though, indeed.

fixed (simple enough) in https://github.com/morpho-org/midnight/pull/753

MathisGD

at the end we won't fix it

Issue L-9: Pending fee can be reduced by repeated self-transfer [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/28

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

Vulnerability Detail

Assume that:

- Time to maturity = 360 days
- Maximum effective fee rate over 360 days = `f = 0.01 * 360/365 = 0.009863` (0.9863%)
- Trading fee = 0%

Scenario A - Single holder

1. Alice (lender/maker) lends and receives `credit = 10,000`, `pendingFee = 10,000 * 0.009863 = 98.63`
2. Protocol collects ~98.63 units at maturity

Scenario B - Self-Transfer at Day 180

1. At day 180, `_updatePosition` is called: `accruedFee = 49.31`, `credit = 9,950.69`, `pendingFee = 49.32`
2. Alice sells her position to Alice2 (controlled account). Alice2 receives `credit = 9,950.69`, `pendingFee = 9,950.69 * 0.009863 * (180/360) = 49.07`
3. Protocol collects 49.31 + 49.07 = ~98.38 units at maturity

Scenario C - Self-Transfer every block (theoretically maximum)

1. Protocol collects = ~98.15 units (`10,000 × (1 − ê(-0.009863))`)

So the maximum gap versus a single holder is about ~0.48 units on 10000 notional units, or roughly `0.48 bp`

Impact

Since the current maximum continuous fee is capped at 1% per year, the worst-case scenario scales approximately as f^2 / 2, which is economically small and generally dominated by trading and gas fees unless the gas or trading fee is zero or negligible. So, this is informational and an observation for awareness.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L742

Tool Used

Manual Review

Recommendation

Ideally, the buyer should inherit the seller's pro-rata remaining `pendingFee` for the portion of credit actually transferred, while only newly originated credit should get a fresh fee initialized from the remaining time to maturity.

If this additional complexity is not worth it, consider documenting the current behavior.

Discussion

MathisGD

this is documented already:

```
/// @dev Because of roundings, trading and continuous fees might charge less than
 ↪  expected, which can become problematic
/// for chains where the gas is cheaper than 1 asset of the loan token.
```

we don't round against the users because you could imagine some attacks against the makers. and we judged this limitations as ok, because it's exploitable only if the gas cost is cheaper than 1 asset, which is not realistic.

so we ack

Issue L-10: Side-effect of ”optimistic” bad debt socialization [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/29

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

Vulnerability Detail

The new design intentionally supports zero-amount liquidation to enable permissionless bad debt realization without any funds, and also ”optimistically” socializes all bad debts right from the start.

However, there are some side effects from this new design. If there is a sudden, temporary price dip that causes borrowers' positions to incur bad debt, they will rush to perform a zero-amount liquidation against their own positions. In this case, part of their debt will be written off at the lender's expense (borrower's debt reduced, but collateral remains untouched). So, if the collateral price recovers later, borrowers will enjoy the upside, while lenders will get slashed.

Having said that, the borrowers still need to ”race” against other liquidators to achieve that in such a scenario.

Impact

Part of the borrower's debts will be written off at the lender's expense without having their collateral liquidated.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L505

Tool Used

Manual Review

Recommendation

With most design choices made, there will always be some trade-off. The new design is optimized to ”optimistically” socialize all bad debts right from the start, rather than the common approach of only recognizing bad debts at the end, when the position is liquidated, which introduces some delay.

If the protocol team decides to ”optimize” further in the current or future version, consider adopting a design somewhere in the middle ground between these two spectrums where bad debts are socialized linearly (e.g., proportional to debt repaid) while the liquidations are ongoing.

Discussion

MathisGD

we ack this issue.

We had it in mind, and think that the tradeoff is good (being able to quickly realize without liquidating, vs what you described).

Note that the way the oracle behaves is a lever to change the behavior of the liquidation and bad debt realization. But there is no magic neither...

Issue L-11: Transfer's comments improvement [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/30

Vulnerability Detail

The protocols utilize the `SafeERC20Lib.safeTransfer()` and `safeTransfer.safeTransferFrom()` functions within the codebase. Within the functions, the implementation will revert if the returned boolean is false OR the transfer operation reverts.

```
File: SafeERC20Lib.sol
09:     function safeTransfer(address token, address to, uint256 value) internal {
10:         require(token.code.length > 0, ErrorsLib.NoCode());
11:
12:         (bool success, bytes memory returndata) =
 ↪  token.call(abi.encodeCall(IERC20.transfer, (to, value)));
13:         require(success, ErrorsLib.TransferReverted());
14:         require(returndata.length == 0 || abi.decode(returndata, (bool)),
 ↪  ErrorsLib.TransferReturnedFalse());
15:     }
```

Following are the related to comments that attempt to document the expected behavior of ERC20 tokens supported by the protocol.

```
File: Midnight.sol
82: /// - It should not revert on `transfer` and `transferFrom` if balances and
 ↪  approvals are right.
83: /// - It should not revert on no-op transfers.
```

```
File: Midnight.sol
93: /// @dev If a token pulled by Midnight reverts on `transferFrom` despite
 ↪  balances and approvals being right, `take`,
94: /// `repay`, `supplyCollateral`, `liquidate`, and `flashLoan` repayment revert
 ↪  when they need to pull that token.
```

```
95: /// @dev If a token sent by Midnight reverts on `transfer` despite balances
↪  being right, `withdraw`,
96: /// `withdrawCollateral`, fee claims, the collateral leg of `liquidate`, and
↪  `flashLoan` revert when they need to send
97: /// that token.
```

The comments correctly documented the transfer operation should be revert, but it does not document that returned boolean should not return false if balances and approvals are right.

```
File: SafeERC20Lib.sol
09:     function safeTransfer(address token, address to, uint256 value) internal {
10:         require(token.code.length > 0, ErrorsLib.NoCode());
11:
12:         (bool success, bytes memory returndata) =
↪  token.call(abi.encodeCall(IERC20.transfer, (to, value)));
13:         require(success, ErrorsLib.TransferReverted());
14:         require(returndata.length == 0 || abi.decode(returndata, (bool)),
↪  ErrorsLib.TransferReturnedFalse());
15:     }
16:
17:     function safeTransferFrom(address token, address from, address to, uint256
↪  value) internal {
18:         require(token.code.length > 0, ErrorsLib.NoCode());
19:
20:         (bool success, bytes memory returndata) =
↪  token.call(abi.encodeCall(IERC20.transferFrom, (from, to, value)));
21:         require(success, ErrorsLib.TransferFromReverted());
22:         require(returndata.length == 0 || abi.decode(returndata, (bool)),
↪  ErrorsLib.TransferFromReturnedFalse());
23:     }
```

Impact

N/A

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/main/morpho-org__midnight/integration/vault-v2/src/libraries/SafeERC20Lib.sol#L9

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/main/morpho-org__midnight/integration/vault-v2/src/libraries/SafeERC20Lib.sol#L17

Tool Used

Manual Review

Recommendation

Consider updating the comments if necessary.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/751

Issue L-12: Potential overflow due to checked multiplications through `mulDivDown/mulDivUp` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/31

Vulnerability Detail

Some features will also break if oracle returns a large value that causes checked multiplications through `mulDivDown/mulDivUp` to overflow and revert.

Although it is unlikely that any assets will be priced so high, for completeness, this behavior should be documented.

```
File: Midnight.sol
85: /// LIVENESS
86: /// @dev If an activated collateral oracle reverts on `price`, `liquidate`,
↪  `isHealthy`, `withdrawCollateral`  when the
87: /// borrower has debt, and `take` whenever the seller still has debt all revert.
88: /// @dev If an activated collateral oracle returns 0 on `price`, `isHealthy`,
↪  `withdrawCollateral` when the borrower has
89: /// debt, `take` whenever the seller still has debt, and `liquidate` with
↪  repaid input all revert.
90: /// @dev If `enterGate.canIncreaseCredit` reverts or returns false, `take`
↪  reverts if the buyer's credit increases.
91: /// @dev If `enterGate.canIncreaseDebt` reverts or returns false, `take`
↪  reverts if the seller's debt increases.
92: /// @dev If `liquidatorGate` reverts or returns false on `canLiquidate`,
↪  `liquidate` reverts.
93: /// @dev If a token pulled by Midnight reverts on `transferFrom` despite
↪  balances and approvals being right, `take`,
94: /// `repay`, `supplyCollateral`, `liquidate`, and `flashLoan` repayment revert
↪  when they need to pull that token.
95: /// @dev If a token sent by Midnight reverts on `transfer` despite balances
↪  being right, `withdraw`,
```

```
96: /// `withdrawCollateral`, fee claims, the collateral leg of `liquidate`, and
 ↪  `flashLoan` revert when they need to send
97: /// that token.
98: /// @dev If a callback reverts, or if a buy/sell callback returns something
 ↪  other than `CALLBACK_SUCCESS`,
99: /// callback-enabled `take`, `repay`, `liquidate`, and `flashLoan` revert.
```

Impact

N/A

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L86

Tool Used

Manual Review

Recommendation

Consider updating the comments if necessary.

Discussion

MathisGD

https://github.com/morpho-org/midnight/pull/752

Issue L-13: Existing fillable orders might break after updating the trading fee [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/32

Vulnerability Detail

For already-created obligations, executing `setObligationTradingFee()` to increase the trading fee might cause orders that are fillable before the update to be unfillable after the update due to underflow when computing the `sellerPrice = offerPrice - tradingFee` because `tradingFee > offerPrice`The similar impact might occur for `setDefaultTradingFee()` too.

Assume the current default fee for a 360-day period is 0.40%. Alice signs a buy order based on the on-chain default 0.40%, which is fillable. However, before `take()` executes Alice's order, the fee setter calls `setDefaultTradingFee` to set the fee to 0.50%. So, when `take()` executes `touchObligation()` to first initialize the obligation, it copies the 0.50% fee to the obligation, and Alice's order is now unfillable due to underflow.

Impact

Existing fillable orders might break after the trading fee is updated.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L171

Tool Used

Manual Review

Recommendation

Consider documenting this behavior so that the fee setter is aware that updating/increasing the trading fee might break existing fillable orders.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/757

Issue L-14: Initiator address is not passed into the `onFlashLoan()` function [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/33

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

Vulnerability Detail

If a user or an external protocol creates a callback contract that handles incoming onFlashLoan requests, the `msg.sender` is always Midnight. The callback contract will usually include a `msg.sender == midnight_address` check to ensure that only authorized protocol can trigger the callback contract.

However, there is no way for the callback contract to determine who is really initiating the request. The callback contract could probably try to decode the `data` variable to fetch the caller, but `data` can be arbitrarily defined by anyone and can be spoofed unless it implements some signature verification on the data, which is much more complicated.

Thus, the `onFlashLoan()` function should pass in the `msg.sender` along with other data so that the callback contract can have direct visibility into the initiator address and implement necessary access controls against the initiator address.

Note 1: It is not uncommon to pass in the initiator address in the `onFlashLoan()` function, as it is defined in the EIP-3156 standard

Note 2: In morpho blue, the `onFlashLoan()` callback will always be executed against the `msg.sender` (caller/initiator), so this is not necessary and not a problem.

Impact

User or external protocol's callback contracts might find it difficult to implement access control against the initiator address.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L653

Tool Used

Manual Review

Recommendation

Consider passing the initiator address when calling `onFlashLoan()` function.

Discussion

MathisGD

fixed by https://github.com/morpho-org/midnight/pull/750

MathisGD

at the end we didn't include it. balancer doesn't send the initiator neither. and it would create a discrepancy with other functions in midnight (for which there are real good reasons not to do it)

Issue L-15: Incumbent lenders are involuntarily rolled from realized cash back into fresh borrower risk [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/34

Vulnerability Detail

In classic bond logic, when a borrower sells a 100 face-value unit for 60, the 40 discount is the risk premium the lender is compensated for locking up funds until maturity and bearing default risk. So, the 40 discount/premium belongs exclusively to the person who supplied the 60 funds that are now at risk.

In the midnight's design, credits are fungible, and all credit holders are treated as valid claimants on the pool's withdrawable cash on a ”first-come-first-served” basis at face value. If someone repays their debt, it will go to the shared cash pool, and any lenders can redeem it. If there is withdrawable cash, the user can buy new issuance at a discount to par and immediately redeem it at par against pooled cash.

So technically, the discount no longer compensates for risk-bearing because the buyer never bears risk. So the risk sort of ”transferred” to the incumbent lenders.

Assume:

- T1 (Someone repay debt): Withdrawable = 100, Alice's credit = 100, Borrower's debt = 0, Bob's credit = 0, Bob's net = 0
- T2 (Bob buy): Withdrawable = 100, Alice's credit = 100, Borrower's debt = 100, Bob's credit = 100, Bob's net = -60
- T3 (Bob redeem): Withdrawable = 0, Alice's credit = 100, Borrower's debt = 100, Bob's credit = 0, Bob's net = +40

So, Alice previously realized cash claim had been involuntarily rolled back into exposure to the new borrower's debt. The 40 discount is supposed to compensate the party whose capital actually funds the new debt and bears maturity/default risk, but this relationship

breaks down as Bob captures the discount, and existing lenders' already-withdrawable cash ends up funding the new borrower exposure.

Note 1: Bob can also do this without supplying his own funds by using the `onBuy()` hook because 100 will be credited to Bob before the `onBuy()` hook is triggered. In the `onBuy()`, he can withdraw 100. When the control is passed back to midnight, midnight will pull 60 from Bob's callback contract, leaving 40 untouched.

Note 2: In the presence of capable sell credit offers, a rational repayer will always atomically use all the `withdrawable` they create to pocket the current market discount fully. For instance, Bob repays 100 of his own debt, and immediately creates 100 withdrawable. In the same transaction, Bob buys 100 units from another borrower's sell offer for, say, 60. He then withdraws 100 at par against the withdrawable bucket he just created, capturing a 40-spread value.

Impact

Incumbent lenders are involuntarily rolled from realized cash back into fresh borrower risk, and are unable to access their repaid funds as funds in the withdrawable bucket are drained. In addition, the spread is captured by the taker.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L250

Tool Used

Manual Review

Recommendation

Consider separating repayment cash entitlements from newly minted credit.

One solution is to track a per-position cash-claim index or claim balance so that only the credit that existed before a repayment can be redeemed. Credit created from fresh

seller debt should start at the current index and must not inherit pre-existing withdrawable.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/872

MathisGD

> Incumbent lenders are involuntarily rolled from realized cash back into fresh borrower risk, and are unable to access their repaid funds as funds in the withdrawable bucket are drained. In addition, the spread is captured by the taker.

Note that lenders subscribed to being lender until the maturity. Not sure that the fact that they can't profit from an early repayment or liquidation can be considered a medium vulnerability. But worth being documented for sure.

Issue L-16: Excessive privilege granted to ratifier [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/35

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

Vulnerability Detail

In the current design, if a user wants to authorize a ratifier, they must add the ratifier to the `isAuthorized` mapping, which effectively grants the ratifier full operator permissions. This means that, in addition to ratifying an offer, the ratifier can also repay, withdraw, borrow, supply, and shuffle sessions.

This deviates from the principle of least privilege. Ideally, the ratifier's scope should only be limited to ratifying the offer.

```
File: Midnight.sol
250:     function take(
..SNIP..
270:         require(isAuthorized[offer.maker][offer.ratifier], "ratifier
↪  unauthorized");
```

Impact

Authorizing a ratifier currently amounts to granting full operator access, so a malicious or compromised ratifier can perform arbitrary account actions apart from usual offer ratification.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L270

Tool Used

Manual Review

Recommendation

Consider creating a separate mapping (e.g., isRatifier) to track authorized ratifiers, ensuring clear separation of duties between ratifiers and operators.

Discussion

peyha

we ack the issue because users will have to check that authorized contracts can only call specific endpoints on midnight. When authorized contracts can act as ratifiers, users will also have to check whether or not the contract can receive `onRatify` calls from midnight. We treat it as a similar analysis and it was documented in #741

Issue L-17: Risk of being unable to or delaying in disabling the compromised operator addresses. [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/36

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

Vulnerability Detail

In the new design, besides the owner, an authorized address can also authorize other addresses.

```
File: Midnight.sol
643:     /// @dev Authorized addresses can authorize other addresses to act on
 ↪  their behalf so it should be used carefully.
644:     function setIsAuthorized(address onBehalf, address authorized, bool
 ↪  newIsAuthorized) external {
645:         require(onBehalf == msg.sender || isAuthorized[onBehalf][msg.sender],
 ↪  "unauthorized");
646:         isAuthorized[onBehalf][authorized] = newIsAuthorized;
647:         emit EventsLib.SetIsAuthorized(msg.sender, onBehalf, authorized,
 ↪  newIsAuthorized);
648:     }
```

The issue is that as long as any authorized address is compromised, it will be difficult for a normal user to disable it. The attacker will likely spawn a large number of authorized addresses, and the user has to go through all the events or mappings to find them all and disable them.

If the user submits the transaction to disable compromised authorized addresses via public mempool, the attacker can also front-run the user to authorize additional addresses. So, the outcome is that the user will either be unable to disable them or will be delayed in doing so, giving the attacker additional time to drain funds from the user's

account.

Impact

Risk of being unable to or delaying in disabling the compromised operator addresses.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/58b07ff3d5e999dcc8ea66978ca683ffb5ebfee0/morpho-org__midnight/src/Midnight.sol#L644

Tool Used

Manual Review

Recommendation

Consider allowing only the user to authorize a new address. In this case, if any of the authorized addresses is compromised, they can easily disable it.

Discussion

MathisGD

As the authorization system is not granular (all or nothing), it's very unlikely to be used to authorize EOAs. Users would rather authorize smart-contracts that statically constrain what they can and cannot do, to achieve granular authorizations. In that context, we think that everything should be authorized. It doesn't add much risk, and it unlocks some use-cases. Note also that this is documented.

We acknowledge this issue.

Issue L-18: Zero amount liquidate won't work if underlying doesn't support zero amount transfer [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/37

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

Summary

`liquidate()` supports `repaidUnits == 0` and `seizedAssets == 0` to realize bad debt without repayment or collateral seizure. However, after accounting updates it always calls both `safeTransfer(collateralToken, msg.sender, seizedAssets)` and `safeTransferFrom(loanToken, msg.sender, address(this), repaidUnits)`. If either token reverts on zero-value transfers, the whole bad-debt realization reverts.

Vulnerability Detail

In Midnight zero-amount liquidations are allowed by `atMostOneNonZero(repaidUnits, seizedAssets)`, and the bad-debt accounting branch can execute before any asset movement. But the function unconditionally performs token transfers even when both amounts are zero. When `seizedAssets == 0` (liquidation was called for realizing bad debt), there still this unconditional transfer, which can block the operation for zero-amount transfer reverting tokens.

Impact

Bad debt may remain unrealized for affected markets. This can block intended loss socialization, keep accounting stale, and interfere with downstream liquidation/withdrawal assumptions.

Code Snippet

Midnight.sol#L565-L622

```
if (repaidUnits > 0 || seizedAssets > 0) {
    ...
}

emit EventsLib.Liquidate(...);

SafeTransferLib.safeTransfer(obligation.collateralParams[collateralIndex].token,
↪  msg.sender, seizedAssets);

if (data.length > 0) {
    ICallbacks(msg.sender)
        .onLiquidate(id, obligation, collateralIndex, seizedAssets, repaidUnits,
        ↪  borrower, data);
}

SafeTransferLib.safeTransferFrom(obligation.loanToken, msg.sender, address(this),
↪  repaidUnits);
```

Tool Used

Manual Review

Recommendation

```
- SafeTransferLib.safeTransfer(obligation.collateralParams[collateralIndex].token,
↪  msg.sender, seizedAssets);
+ if (seizedAssets > 0) {
+     SafeTransferLib.safeTransfer(obligation.collateralParams[collateralIndex].tok
↪  en, msg.sender, seizedAssets);
+ }

- SafeTransferLib.safeTransferFrom(obligation.loanToken, msg.sender, address(this),
↪  repaidUnits);
```

```
+ if (repaidUnits > 0) {
+     SafeTransferLib.safeTransferFrom(obligation.loanToken, msg.sender,
↪  address(this), repaidUnits);
+ }
```

Discussion

MathisGD

it's part of the ERC20 spec, and we documented that tokens must be ERC20 compliant (except specific exceptions, that don't include this)

> Note Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.

thus we acknowledge

Issue L-19: The absence of actor binding lets attackers drain third-party callback funds across maker-buyer, taker-buyer, and seller orders [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/38

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

Summary

The callback examples do not bind callback authority to the mapped economic actor (`buyer/seller`). `take()` accepts callback addresses from maker-signed `offer.callback` and caller-supplied `takerCallback`, then uses callbacks in value-moving paths.

Vulnerability Detail

With permissive integrations that only trust `msg.sender == Midnight` (or obligation id), an attacker can:

1. drain loan tokens through maker-side buyer callback,
2. drain loan tokens through taker-side buyer callback,
3. drain collateral through seller callback legs,
4. drain callback-held balances by making `flashLoan()` invoke an arbitrary third-party callback with `assets = 0`.

Impact

Direct theft up to callback-funded balances:

- loan-token callback drain on buyer legs,

- collateral callback drain on seller legs.
- callback-held balance drain on the flash-loan leg.

Code Snippet

There is the reference pattern that validates only `msg.sender == Midnight`, e.g. without binding the `buyer`:

TakeTest.sol#L1670-L1678

```
function onBuy(bytes32 id, Obligation memory obligation, address, uint256
↪  buyerAssets, uint256, bytes memory data)
    external returns (bytes32)
{
    require(id == IdLib.toId(obligation, block.chainid, msg.sender), "wrong id");
>>  ERC20(obligation.loanToken).approve(msg.sender, buyerAssets);
    return CALLBACK_SUCCESS;
}
```

For sell offers, `take()` maps the caller-supplied `takerCallback` to `buyerCallback` and then uses it as the loan-token payer. A taker can point `takerCallback` at any such permissive callback, and Midnight pulls loan tokens from the victim to fund the attacker's operation. I.e. `onBuy()` without binding `buyer` is exploitable. Similar pattern can be seen in the mirror callback: `onBuy` does not bind `buyer`, `onSell` does not bind `seller`.

Tool Used

Manual Review

Recommendation

Consider either documenting the checks as callback development guidance or Midnight's `take()` side can tighten up the rules, e.g.:

```
+ require(
+    takerCallback == address(0)
+        || takerCallback == taker
```

```
+        || isAuthorized[taker][takerCallback],
+    "taker callback unauthorized"
+ );
```

Discussion

peyha

We ack this, makers/taker should check the behavior of the callback before signing an offer/a take transaction just like they should check the receiver address (which is pretty standard behavior in DeFi).

Issue L-20: Zero-unit `take()` can replay callbacks and emit zero-value `Take` events [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/39

Summary

`take(0, ...)` passes the normal offer, session, proof, and ratifier checks. It computes `buyerAssets == 0` and `sellerAssets == 0`, does not advance `consumed`, still emits a `Take` event, and still invokes the buyer and seller callbacks with zero amounts. Because `newConsumed += 0`, this also means a fully consumed offer can still be hit with a zero-unit call.

Vulnerability Detail

It can be used for spam griefing, e.g. if it is known that maker's callback does something heavy off-chain despite zero amount, or just to spam Take events when it's known that an entity processing them can be overloaded this way.

Impact

The impact is off-chain DOS against third-party systems: the attacker might want to pay for the gas (on L2 or during low activity mainnet period) knowing that some further off-chain, derivative, operations on the side of a target third-party entity might be comparably expensive, and having some material interests in strangling that entity during some period as a part of some DDOS operation.

Code Snippet

Zero units produce zero buyer and seller assets, and consumed accounting does not advance.

Midnight.sol#L301-L318

```
uint256 offerPrice = TickLib.tickToPrice(offer.tick);
uint256 timeToMaturity = UtilsLib.zeroFloorSub(offer.obligation.maturity,
↪  block.timestamp);
uint256 _tradingFee = tradingFee(id, timeToMaturity);
uint256 sellerPrice = offer.buy ? offerPrice - _tradingFee : offerPrice;
uint256 buyerPrice = sellerPrice + _tradingFee;
uint256 buyerAssets = offer.buy ? units.mulDivDown(buyerPrice, WAD) :
↪  units.mulDivUp(buyerPrice, WAD);
uint256 sellerAssets = offer.buy ? units.mulDivDown(sellerPrice, WAD) :
↪  units.mulDivUp(sellerPrice, WAD);

...

newConsumed = consumed[offer.maker][offer.group] += units;
require(newConsumed <= offer.maxUnits, "consumed units");
```

`Take` event carrying zero amounts is emitted:

Midnight.sol#L363-L379

```
emit EventsLib.Take(
    ...
);
```

Callbacks are invoked with zero amounts and zero units:

Midnight.sol#L381-L400

```
if (buyerCallback != address(0)) {
    require(
        ICallbacks(buyerCallback).onBuy(id, offer.obligation, buyer, buyerAssets,
        ↪  units, buyerCallbackData)
            == CALLBACK_SUCCESS,
        "invalid callback"
    );
}

...
```

```
if (sellerCallback != address(0)) {
    require(
        ICallbacks(sellerCallback).onSell(id, offer.obligation, seller,
        ↪  sellerAssets, units, sellerCallbackData)
            == CALLBACK_SUCCESS,
        "invalid callback"
    );
}
```

Tool Used

Manual Review

Recommendation

Consider documenting to warn integrators against using spammable logic in callbacks and event processing, or simply rejecting zero-unit fills as they don't carry a separate business sense, e.g.:

```
 function take(
     uint256 units,
     address taker,
     ...
 ) external returns (uint256, uint256, uint256) {
+    require(units > 0, "zero units");
     bytes32 id = touchObligation(offer.obligation);
     ObligationState storage _obligationState = obligationState[id];
     ...
 }
```

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/761 (added a comment)

Issue L-21: Fee slippage and stale credit documentation [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/41

Summary

1. `take()` docstring warns that the taker may not get the expected price if trading fees were changed just before inclusion, but it does not point readers to the existing ways to bound that slippage.
2. `creditOf()` and the internal `hasCredit()` helper read stored credit before `_updatePosition()` applies the latest loss-index slash and continuous-fee accrual, so they can overestimate a user's current effective credit.

Both can be important for integrators and operators who build routing, quoting, callbacks, or monitoring around the core contract.

Vulnerability Detail

1. Fee-slippage in `take()`

`take()` already documents that a taker might not receive the price they expected if the trading fee was just changed. This is because `buyerAssets` and `sellerAssets` are derived at execution time from the current per-obligation trading fee, which can be changed atomically by `feeSetter`. I.e. quotes may be prepared under one fee configuration, the protocol fee setter may update trading fees before the transaction lands, which then settles with different buyer/seller asset amounts than the caller expected.

2. `creditOf()` / `hasCredit()` can be stale before `_updatePosition()`

`creditOf()` returns raw stored credit, and `hasCredit()` checks whether raw stored credit is nonzero. Neither applies the current `lossIndex` nor the pending continuous-fee accrual first.

The contract's canonical fresh state is computed by `updatePositionView()`, which slashes

credit against the latest market `lossIndex`, adjusts pending fee after the slash, accrues additional fee from `lastAccrual` up to `min(block.timestamp, obligation.maturity)`, returns the post-slash, post-accrual credit that `_updatePosition()` would write.

So, before `_updatePosition()` runs, a position may still report positive stored credit through `creditOf()` and `hasCredit()` even though its current effective credit is already lower. In extreme cases after a full slash, the stale stored value can be nonzero while the fresh value is zero.

Impact

1. Fee-slippage: direct `take()` callers can receive worse-than-expected execution if the trading fee changes between quoting and inclusion, while documentation that does not point to min/max or callback-based controls makes misuse more likely.
2. Stale-credit: integrators may overestimate currently effective credit if they read `creditOf()` without updating the position first, which can lead to incorrect UI balances, routing assumptions, or monitoring conclusions.

Code Snippet

1. Fee-slippage and existing controls:

Midnight.sol#L242-L245

```
/// @dev Same function used to buy and sell.
/// @dev If one wants to match two offers without taking a position, they can batch
 ↪  take them and not have a
/// position at the end.
/// @dev The taker might not get the price they expected if the trading fee was
 ↪  just changed.
```

TakeBundler.sol#L26-L36

```
function bundleTakeUnits(
    Midnight midnight,
    uint256 targetUnits,
    address taker,
    address receiverIfTakerIsSeller,
```

```
    Take[] calldata takes,
    uint256 minBuyerAssets,
    uint256 maxBuyerAssets,
    uint256 minSellerAssets,
    uint256 maxSellerAssets
) external {
```

TakeBundler.sol#L62-L66

```
require(totalFilledUnits == targetUnits, "insufficient liquidity");
require(totalBuyerAssets >= minBuyerAssets, "buyer assets below min");
require(totalBuyerAssets <= maxBuyerAssets, "buyer assets above max");
require(totalSellerAssets >= minSellerAssets, "seller assets below min");
require(totalSellerAssets <= maxSellerAssets, "seller assets above max");
```

2. Stale `creditOf()` / `hasCredit()`:

Midnight.sol#L731-L742

```
function _updatePosition(Obligation memory obligation, bytes32 id, address user)
↪  internal {
    Position storage _position = position[id][user];
    (uint128 newCredit, uint128 newPendingFee, uint128 accruedFee) =
    ↪  updatePositionView(obligation, id, user);

    ...

    _position.credit = newCredit;
    _position.lossIndex = obligationState[id].lossIndex;
    _position.pendingFee = newPendingFee;
    _position.lastAccrual = uint128(block.timestamp);
    obligationState[id].continuousFeeCredit += UtilsLib.toUint128(accruedFee);
```

Midnight.sol#L747-L749

```
function hasCredit(bytes32 id, address user) internal view returns (bool) {
    return position[id][user].credit > 0;
}
```

Midnight.sol#L777-L779

```
function creditOf(bytes32 id, address user) external view returns (uint256) {
    return position[id][user].credit;
}
```

Tool Used

Manual Review

Recommendation

Consider documenting both caveats explicitly:

1. For trading-fee slippage consider expanding the `take()` doc comment to point users to `TakeBundler` min/max parameters as the built-in slippage guard, mentioning that callback-based integrations can also revert after seeing the computed execution amounts if they are outside the caller expectations, noting that direct `take()` users who do not use those controls accept fee-change execution risk. I.e. point to the existing control surfaces:

- `TakeBundler.bundleTakeUnits()` already exposes `minBuyerAssets`, `maxBuyerAssets`, `minSellerAssets`, and `maxSellerAssets`,
- direct `take()` users can also use callbacks as a last-mile assertion point, as the callback receives the computed execution amounts and can revert the whole transaction if they are unacceptable.

2. For stale credit views:

- document that `creditOf()` and `hasCredit()` reflect stored state, not necessarily fresh post-slash / post-accrual state,
- make clear that lazy accrual and loss application are reconciled on `_updatePosition()`, so raw storage views can temporarily overestimate spendable credit.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/762 (except for hasCredit as it's internal)

Issue L-22: `proof.length` can be used instead of`height` in `EcrecoverRatifier.isRatified` [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/43

Summary

`EcrecoverRatifier.isRatified` decodes `height` from the ratifier data and folds it into the EIP-712 typehash. For every tree the protocol accepts under `OfferTree(Offer[2]...[2] offerTree)`, so `height` is always equal to `proof.length`.

Detail

`HashLib.offerTreeTypeHash` only returns typehashes for `Offer[2]^height` balanced binary trees, so the height is the length of the proof.

Impact

Optimization

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/main/morpho-org__midnight/src/ratifiers/EcrecoverRatifier.sol#L36-L40

```
(Signature memory sig, uint256 height, bytes32 root, bytes32[] memory proof) =
    abi.decode(ratifierData, (Signature, uint256, bytes32, bytes32[]));
require(HashLib.isLeaf(root, HashLib.hashOffer(offer), proof), InvalidProof());
require(!isRootCanceled[offer.maker][root], RootCanceled());
bytes32 structHash = keccak256(abi.encode(HashLib.offerTreeTypeHash(height), root));
```

Tool Used

Manual Review

Recommendation

Use `proof.length` instead of height.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/883

Issue L-23: TickLib.tickToPrice(0) = 0 is an unguarded state [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/44

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

Summary

`TickLib` operates on `(0, WAD)` and the upper bound is enforced in `tickToPrice`'s denominator construction and `priceToTick`'s `require(price <= 1e18)`. The lower bound has no equivalent guard: at `tick = 0`, the `4.988e11` value rounds down via `PRICE_ROUNDING_STEP = 1e12` to `0`. Tick 0 is the unique tick that collapses to zero, tick 1 produces `1e12`. Every consumer of `tickToPrice` either divides by the result, subtracts it from `tradingFee`, or feeds it into arithmetic that breaks when it equals zero.

As a most severe sub-case for a sell offer at tick 0 the take math becomes `sellerPrice = 0`, `buyerPrice = tradingFee`, `sellerAssets = 0`, `buyerAssets = units · tradingFee / WAD`. Through `MidnightBundles.buyWithAssetsTargetAndWithdrawCollateral()`, a taker buys position credit at the inverse trading fee rate `WAD / tradingFee`, between 200× (max 360-day fee) and 71428× (max 1-day fee) per loan-token spent. The extraction tops out at the maker's `isHealthy()` bounded debt capacity.

Vulnerability Detail

For `tick = 0`: `wExp(ln(1.005) * 2910)  2.005e24`, outer division produces `4.988e11`, `divHalfDownUnchecked(4.988e11, 1e12) = 0`, then `0 * 1e12 = 0`:

TickLib.sol L42-L50:

```
function tickToPrice(uint256 tick) internal pure returns (uint256) {
    require(tick <= MAX_TICK, TickOutOfRange());
    unchecked {
        return uint256(1e36)
                .divHalfDownUnchecked(1e18 + wExp(LN_ONE_PLUS_DELTA *
                ↪  (int256(MAX_TICK / 2) - int256(tick))))
```

```
                .divHalfDownUnchecked(PRICE_ROUNDING_STEP) * PRICE_ROUNDING_STEP;
    }
}
```

`tickToPrice()` enforces only `tick <= MAX_TICK`. `priceToTick()` enforces only `price <= 1e18`. A maker can construct an offer with `tick = 0` (no validation rejects it), and `priceToTick(0, spacing)` returns `tick = 0` (the binary search starts at `low = 0`, never advances when `tickToPrice(0) < price` is false because both sides are 0). No economic reason to use tick 0 deliberately, but uninitialized defaults and off-by-one tick math can be the accidental paths.

Downstream:

Path	Tick-0 offer side	Consequence
`Midnight.take`	buy	`sellerPrice = 0 − tradingFee`
`Midnight.take`	sell	`sellerPrice = 0`, `buyerPrice =`
`TakeAmountsLib.sellerAssetsToUnits`	sell	`sellerPrice = 0` ⇒ `mulDivDown(`
`TakeAmountsLib.buyerAssetsToUnits`	buy	`sellerPrice = 0 − tradingFee`
`TakeAmountsLib.buyerAssetsToUnits`	sell, `tradingFee = 0`	`buyerPrice = 0` ⇒ div-by-zero.
`TakeAmountsLib.buyerAssetsToUnits`	sell, `tradingFee > 0`	Returns `target · WAD / tradin`
`ConsumableUnitsLib.consumableUnits`	any, `maxAssets` regime	Routes into one of the reverting
`MidnightBundles` all four take entry points	any, `maxAssets` regime	`consumableUnits` sits outside th

Particularly, for the most severe case of `TakeAmountsLib.buyerAssetsToUnits()`, if the following takes place:

1. Sell offer at `tick = 0`.
2. `maxUnits > 0`, `maxAssets == 0`. Under `maxAssets`, `consumableUnits` reverts and the bundler precompute aborts.

3. `tradingFee > 0` for the market's maturity bucket. With `tradingFee == 0`, `buyerAssetsToUnits` div-by-zero.

Then, in `buyWithAssetsTargetAndWithdrawCollateral`'s loop, with sell offer at tick 0:

MidnightBundles.sol L198-L200: inverse call. TakeAmountsLib.sol L29: inflated divide. Midnight.sol L354-L355: forward math. Midnight.sol L362: consumed += units. Midnight.sol L454-L458: isHealthy cap.

```
buyerAssetsToUnits(remaining)  =  remaining · WAD / tradingFee     ← inflated
take(units)  produces:
  buyerAssets = floor(units · tradingFee / WAD)    remaining  (round-trips)
  sellerAssets = ceil(units · 0 / WAD)            = 0
  buyerCreditIncrease = units − buyer.debt         units (taker credit)
  sellerDebtIncrease  = units − seller.credit      units (maker debt)
  consumed[maker][group] += units                 (maxUnits regime)
require(isHealthy(seller))                         ← caps units per take to the
↪  maker's collateral capacity
```

- `isHealthy` caps per-take to the maker's collateral capacity – not to a no-loss baseline. The whole LLTV-bounded amount is takeable.
- `require(buyerPrice <= WAD)` (TakeAmountsLib) – `buyerPrice = tradingFee << WAD`, passes.
- `reduceOnly`, `enterGate.canIncreaseDebt(seller)`, would block if the maker had configured them. So, this case requires a maker who mis-set `tick = 0` and didn't set these guards.

Net per offer: maker absorbs `~collateral · LLTV` units of debt, taker pockets the same in credit (claimable at maturity), taker pays `units · tradingFee / WAD` to the protocol (trading-fee book), nothing to the maker. Conservation: taker `+units·(1−tradingFee/WAD)`, maker `−units`, protocol `+units·tradingFee/WAD`.

Extraction ratios:

Maturity bucket	Max `tradingFee`	Taker credit per loan token paid
< 1 day	`1.4e13`	71,428×

Maturity bucket	Max `tradingFee`	Taker credit per loan token paid
< 7 day	`9.8e13`	10,204×
< 30 day	`4.17e14`	2,398×
< 90 day	`1.25e15`	800×
< 180 day	`2.5e15`	400×
≥ 360 day	`5e15`	200×

The taker pays trading-fee fraction of the value gained, the protocol captures that fee, the maker loses the underlying.

Code Snippet

TickLib.sol L54-L66, `priceToTick` with no `price > 0` floor:

```
function priceToTick(uint256 price, uint256 spacing) internal pure returns
↪  (uint256) {
    require(price <= 1e18, PriceGreaterThanOne());
    uint256 low = 0;
    uint256 high = MAX_TICK;
    while (low != high) {
        unchecked {
            uint256 mid = (low + high) / 2;
            if (tickToPrice(mid) < price) low = mid + 1;
            else high = mid;
        }
    }
    return (low + spacing - 1) / spacing * spacing;
}
```

Midnight.sol L352, `sellerPrice = offerPrice - _tradingFee` can underflow for buy offer at tick 0:

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

Midnight.sol L354-L355, sell offer at tick 0 produces `sellerAssets = 0`:

```
uint256 buyerAssets = offer.buy ? units.mulDivDown(buyerPrice, WAD) :
↪  units.mulDivUp(buyerPrice, WAD);
uint256 sellerAssets = offer.buy ? units.mulDivDown(sellerPrice, WAD) :
↪  units.mulDivUp(sellerPrice, WAD);
```

TakeAmountsLib.sol L17-L47, revert/inflation:

```
function buyerAssetsToUnits(address midnight, bytes32 id, Offer memory offer,
↪  uint256 targetBuyerAssets)
    internal view returns (uint256)
{
    uint256 offerPrice = TickLib.tickToPrice(offer.tick);
    uint256 tradingFee =
        IMidnight(midnight).tradingFee(id,
        ↪  UtilsLib.zeroFloorSub(offer.market.maturity, block.timestamp));
    uint256 sellerPrice = offer.buy ? offerPrice - tradingFee : offerPrice;
    uint256 buyerPrice = sellerPrice + tradingFee;
    require(buyerPrice <= WAD, TickLib.PriceGreaterThanOne());
    return offer.buy ? targetBuyerAssets.mulDivUp(WAD, buyerPrice) :
    ↪  targetBuyerAssets.mulDivDown(WAD, buyerPrice);
}

function sellerAssetsToUnits(address midnight, bytes32 id, Offer memory offer,
↪  uint256 targetSellerAssets)
    internal view returns (uint256)
{
    uint256 offerPrice = TickLib.tickToPrice(offer.tick);
    uint256 tradingFee =
        IMidnight(midnight).tradingFee(id,
        ↪  UtilsLib.zeroFloorSub(offer.market.maturity, block.timestamp));
    uint256 sellerPrice = offer.buy ? offerPrice - tradingFee : offerPrice;
    return
```

```
        offer.buy ? targetSellerAssets.mulDivUp(WAD, sellerPrice) :
        ↪  targetSellerAssets.mulDivDown(WAD, sellerPrice);
}
```

ConsumableUnitsLib.sol L19-L21, propagation into the bundler precompute:

```
return TakeAmountsLib.buyerAssetsToUnits(midnight, id, offer,
↪  offer.maxAssets.zeroFloorSub(consumed));
} else {
return TakeAmountsLib.sellerAssetsToUnits(midnight, id, offer,
↪  offer.maxAssets.zeroFloorSub(consumed));
```

Tool Used

Manual Review

Recommendation

Consider rejecting tick 0 at the library boundary, e.g.:

TickLib.sol L43:

```
- require(tick <= MAX_TICK, TickOutOfRange());
+ require(tick > 0 && tick <= MAX_TICK, TickOutOfRange());
```

Symmetric guard on `priceToTick` at TickLib.sol L55:

```
- require(price <= 1e18, PriceGreaterThanOne());
+ require(price > 0 && price <= 1e18, PriceGreaterThanOne());
```

Discussion

MathisGD

we acknowledge the issue. there is no particular discontinuity, a maker making at a very low tick / very low price will indeed loose a lot.

Issue L-24: Receiver and transaction initiator addresses not passed into the `onLiquidate()` function [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/45

Vulnerability Detail

In the previous version of Midnight, the callback and receiver were always the transaction initiator (`msg.sender`). This was hardcoded in the previous Midnight implementation. In this case, the callback contract that processes the `onLiquidate()` always knows who the transaction initiator and receiver are.

If the callback contract wants to implement any logic or access control that involves the receiver and the transaction initiator, it can be easily derived from (`address(this)`). Since the receiver and transaction initiator are always equal to the callback address, the callback contract can always fully trust them.

However, in the new design, the callback and receiver can be arbitrary addresses, and there is no guarantee that the receiver and the transaction initiator are always trusted. So, there is no way for the callback contract to reliably derive the receiver and transaction initiator if needed.

As such, anyone can call the liquidate function and point to an arbitrary callback contract to have it pay for the liquidation while having the collaterals claimed forwarded to their own wallet (receiver == their wallet). From the callback contract's perspective, there is no way for it to implement decision logic that involves the collateral receiver.

Impact

External protocol's callback contracts might find it difficult to implement access control or implement certain decision logic that involves the receiver and the transaction initiator.

Code Snippet

https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/blob/98a0c16d4b7ff4d401182ea1f990e3710567e063/Midnight.sol#L678

Tool Used

Manual Review

Recommendation

Consider passing in the receiver and transaction initiator addresses to `onLiquidate()` to provide more flexibility.

Discussion

MathisGD

fixed in https://github.com/morpho-org/midnight/pull/894

Issue L-25: Liquidator callbacks cannot gate exposure, so extra allowance can be exploited [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-morpho-midnight-apr-6th-2026/issues/46

Summary

`Midnight.liquidate` sends `seizedAssets` to `receiver`, invokes `ILiquidateCallback.onLiquidate`, then pulls `repaidUnits` from `payer` via `safeTransferFrom`. Any liquidator that gates `CALLBACK_SUCCESS` on its own collateral-balance delta can be tricked: a third party that, for example, supplies the liquidator with a small amount of the seized collateral token directly, off-Midnight, then calls `Midnight.liquidate(..., receiver = attacker, callback = liquidator)`, forces the liquidator's callback to sign off on a payment it cannot refuse afterwards. The exact atomic allowance looks to be the only way to control for that, i.e. any extra allowance can be directly stolen from such a liquidator.

Vulnerability Detail

Midnight.sol L673-L684:

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

if (callback != address(0)) {
    require(
        ILiquidateCallback(callback)
            .onLiquidate(id, market, collateralIndex, seizedAssets, repaidUnits,
            ↪  borrower, data)
        == CALLBACK_SUCCESS,
        WrongLiquidateCallbackReturnValue()
    );
}
```

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

A successful return commits `payer` to the upcoming transfer. The callback's only on-chain levers are: return non-`CALLBACK_SUCCESS`, revoke the loanToken allowance, or drop balance below `repaidUnits`; each reverts the whole tx, none distinguishes attacker-routed call from liquidator's own batch.

Allowance this way is the only place exposure can be bounded: per-call allowance of exactly `repaidUnits`, granted immediately before the liquidator's own `Midnight.liquidate` and revoked right afterwards controls for the surface.

Multi-liquidation batches can widen the attack window proportionally: on market down-moves where many liquidator contracts are active, the attack can be enabled by increasing the current allowance to be the sum of on-going liquidations' needs, so the attacker can wait for that programmatically.

The same callback-before-pull ordering applies to `onRepay` (L496-L502) and `onFlashLoan` (L712-L718).

Tool Used

Manual Review

Recommendation

Consider describing that on `Midnight.liquidate` (and the matching positions for `repay` and `flashLoan`), e.g.:

```
+ /// @dev `onLiquidate` runs before the loanToken `safeTransferFrom`. A
↪  `CALLBACK_SUCCESS`
+ /// return commits the payer to the upcoming pull. Callback implementations
↪  should scope
+ /// `loanToken` allowance to the exact `repaidUnits` of the call they initiate.
```

Discussion

MathisGD

indirectly fixed by https://github.com/morpho-org/midnight/pull/894

the callback can directly check the liquidator

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.