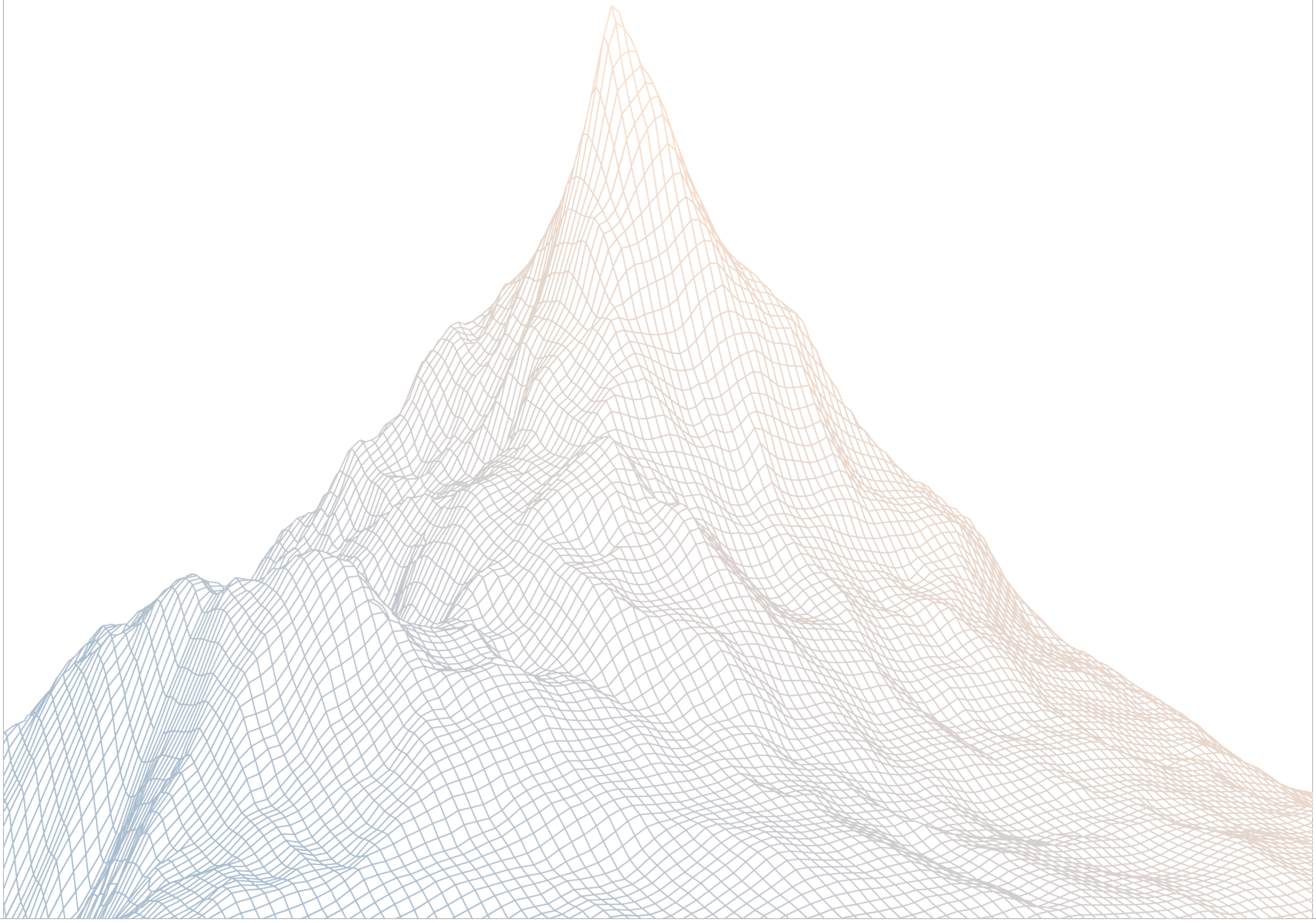
Zenith

Monorail

Smart Contract
Security Assessment

VERSION 1.1



AUDIT DATES: September 24th to September 29th, 2025

AUDITED BY: ether_sky
kaden

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Monorail

Instantly trade anything across Monad. Monorail connects you to every exchange on Monad to give you the absolute best price for your trade. Access 71 454 tokens across 16 exchanges through a single platform.

2.2 Scope

The engagement involved a review of the following targets:

Target	contracts
Repository	https://github.com/monorail-xyz/contracts
Commit Hash	`dcac3bfe8c712b6cbfabaaa7c2068ed31d743287`
Files	`aggregation_v4/src/MonorailAggregator.sol`

Target	Monorails Mitigation Review
Repository	https://github.com/monorail-xyz/contracts
Commit Hash	`f0667c1d9ceaffd0ae19339caa08ccbf73a55338`
Files	`Changes in the latest source code version`

2.3 Audit Timeline

September 24, 2025	Audit start
September 29, 2025	Audit end
October 3, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	2
Informational	13
Total Issues	**16**

3 Findings Summary

ID	Description	Status
M-1	Calls to aggregatePermit2 can be griefed	Resolved
L-1	Fee recipients can DoS native tokenOut trades	Acknowledged
L-2	Approval to arbitrary token spender	Resolved
I-1	tokenIn may be a different token from trades[0].tokenIn	Resolved
I-2	Remaining tokens may stay in contract after multi-leg trades	Acknowledged
I-3	Fee calculation uses round-down instead of round-up	Resolved
I-4	There is a missing parameter documentation in function comments	Resolved
I-5	There is a minor typo and case inconsistency in the comments of the _swapCloberOrderbook function	Resolved
I-6	There is a documentation mismatch for aggregatePermit2 token input	Resolved
I-7	Inconsistent error handling between custom errors and string reverts	Resolved
I-8	The _calculateAmounts function contains a redundant condition check	Resolved
I-9	The initialization of defaultProtocolFeeBps is unnecessary	Resolved
I-10	Some validations are not implemented in the setReferrerFeeBps function	Resolved
I-11	Incorrect comment	Resolved
I-12	Missing NatSpec comment	Resolved
I-13	Trade weight logic may lead to unexpected behavior	Acknowledged

4 Findings

4.1 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] Calls to `aggregatePermit2` can be griefed

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol#L464

Description:

`MonorailAggregator.aggregatePermit2` includes a Permit2 allowance `permit` before using that allowance to transfer the tokens in from the `msg.sender`:

```
permit2.permit(msg.sender, permit, signature);
permit2.transferFrom(
    msg.sender,
    address(this),
    amountIn.toUint160(),
    tokenIn
);
```

Since the stored permit `nonce` will be incremented, `permit2.permit` can only be called once with a given `permit` and subsequent calls with the same `permit/signature` will revert. As a result, since anyone can call `permit` on the Permit2 contract with the same parameters, a griefer can do so and cause the `aggregatePermit2` call to revert unexpectedly.

Recommendations:

This can be resolved by wrapping the `permit2.permit` call in a `try/catch` block, continuing execution regardless of whether the `permit` call reverts. Alternatively, this can be resolved more efficiently by instead using `Permit2.permitTransferFrom` which validates that the `msg.sender` is the `spender`, preventing a griefing attack such as this one.

Monorail: Resolved in @54be250....

Zenith: Verified, resolved via `try/catch` mechanism as recommended.

4.2 Low Risk

A total of 2 low risk findings were identified.

[L-1] Fee recipients can DoS native `tokenOut` trades

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Acknowledged	`LIKELIHOOD`: Low

Target

- MonorailAggregator.sol#L904-L918

Description:

In `MonorailAggregator._transferTokens`, we transfer the `destinationAmount` as well as any fees to be paid. If the `tokenOut` is the native token, we perform these transfers by making a `call` to the recipients with the amount as the `value`. In this case, it's possible for either the `referrerDetails.receiver` or the `protocolFeeReceiver` to intentionally revert the call, effectively censoring users.

Recommendations:

Ensure that the `protocolFeeReceiver` and any approved `referrer` is either trusted or an immutable smart contract which cannot possibly revert incoming native token transfers.

Monorail: Acknowledged. We currently talk to every team requesting fee sharing and manually add their referrer details. Even though it would be detrimental only to their own usage of our contract, we'll keep it in mind and ensure they are aware of this potential issue as well.

[L-2] Approval to arbitrary token spender

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol#L1244

Description:

In `MonorailAggregator._swapCrystalOrderbook`, we approve the `tokenIn` to the provided `market` so that the `market` contract can pull the tokens while executing the swap:

```
(address market, address referrer) = abi.decode(
    extraParams,
    (address, address)
);

// Crystal requires approvals to the market directly
_approveToken(tokenIn, market, amountIn);

ICrystalRouter tradeRouter = ICrystalRouter(router);
tradeRouter.swapExactTokensForTokens(
    amountIn,
    minAmountOut,
    path,
    address(this),
    deadline,
    referrer
);
```

Note that the `market` is an arbitrary address provided by the caller which is never validated to be a legitimate Crystal market. As such, an attacker could provide a contract they control as the `market` to make the `MonorailAggregator` contract max approve any token, requiring only that a valid Crystal swap is executed on a market that has been previously approved.

In practice, this does not currently pose a significant threat as the `MonorailAggregator` contract is not intended to hold tokens. Additionally, it's also possible to skim tokens left in the contract simply by providing those tokens as the `tokenOut` parameter. However, it's recommended to resolve this regardless.

Recommendations:

Use the `CrystalRouter.getMarket` public getter to retrieve the market address for the given `tokenIn/tokenOut` pair, taking care to ensure that the WETH address is provided in place of ETH as is done in `CrystalRouter.exactInputSwap`:

```
// snippet from CrystalRouter.exactInputSwap demonstrating safe market
    retrieval
address asset0 = path[i] == ETH ? WETH : path[i];
address asset1 = path[i+1] == ETH ? WETH : path[i+1];
address market = getMarket[asset0][asset1];
```

Monorail: Resolved with @f93070b7410...

Zenith: Verified.

4.3 Informational

A total of 13 informational findings were identified.

[I-1] `tokenIn` may be a different token from `trades[0].tokenIn`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol#L635-L650

Description:

In `MonorailAggregator._executeCore`, we set the `tokenIn` as the `currentTokenIn`, and for each trade where `currentTokenIn ≠ tradeTokenIn`, we update `currentTokenIn` and `tokenInBalance`:

```
address currentTokenIn = tokenIn;
uint256 tokenInBalance = amountIn;
for (uint256 i = 0; i < totalTrades; i++) {
    // Load all trade data onto the stack once per loop.
    Trade calldata trade = trades[i];
    address router = trade.router;
    address tradeTokenIn = trade.tokenIn;
    address tradeTokenOut = trade.tokenOut;

    // Get the balance of the input token for the current trade leg.
    // This balance is the result of the previous trade or the initial
    deposit.
    if (currentTokenIn ≠ tradeTokenIn) {
        // If the current token in is not set or different, update it.
        tokenInBalance = _getBalance(tradeTokenIn);
        currentTokenIn = tradeTokenIn;
    }
```

One thing to consider here is that the `tokenIn` is not validated to be the same address as `trades[0].tokenIn`. Then in the first loop iteration, we will immediately skip to `trades[0].tokenIn`, ignoring the `tokenIn` altogether. By manually transferring `trades[0].tokenIn` tokens into the contract, the trade will still be possible.

This allows for the ability to bypass validation logic performed on the `tokenIn`, and it may also be used to cause an unexpected event to be emitted.

Recommendations:

Revert in `_executeCore` in case `tokenIn ≠ trades[0].tokenIn`.

Monorail: Resolved in @51c611330b4....

Zenith: Verified.

[I-2] Remaining tokens may stay in contract after multi-leg trades

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

The `_executeCore` function executes multiple `trades` in a single transaction, consuming user-supplied tokens.

- MonorailAggregator.sol#L609

```
function _executeCore(
    address sender,
    address destination,
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 minAmountOut,
    uint256 deadline,
    uint64 referrer,
    uint64 quote,
    Trade[] calldata trades
) private {
    ...
    for (uint256 i = 0; i < totalTrades; i++) {
        // Load all trade data onto the stack once per loop.
        ...
    }
    ...
}
```

If the user provides `trade` data that does not consume the entire `balance` of `input tokens`, or if a `trade outputs` tokens that are not fully `swapped` in subsequent legs, some tokens may remain in the contract.

Currently, there is no mechanism to return leftover tokens to the user after all `trades` complete. This can result in user funds being unintentionally retained in the contract.

Recommendations

After executing all `trades`, return any remaining tokens to the user.

Monorail: Acknowleded. I do agree it would be good practice to check if any of the original input wasn't spent and send that back, but again, the gas penalty might come into play if they used 100% of their input token holdings. For now we'll acknowledge this as an improvement to be made. Once we have clarity on the gas changes we can look at this again.

[I-3] Fee calculation uses round-down instead of round-up

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

In the `_calculateAmounts` function, `fees` are calculated like this.

- MonorailAggregator.sol#L843

```
function _calculateAmounts(
    address tokenOut,
    uint64 referrer
)
    private
    view
    returns (
        uint256 destinationAmount,
        FeeDetails memory referrerDetails,
        uint256 protocolFeeAmount
    )
{
    uint256 finalAmountOut = _getBalance(tokenOut);

    // Load the referrer's fee info
    ReferrerFeeInfo memory feeInfo = referrerFees[referrer];

    // A valid, active referrer is provided
    if (feeInfo.receiver ≠ address(0) && feeInfo.totalFeeBps > 0) {
        if (feeInfo.totalFeeBps > 0) {
@>          uint256 totalFeeAmount = (finalAmountOut *
                feeInfo.totalFeeBps) / BASIS_POINTS;
        }
    }
}
```

This effectively `rounds down` the `fee` amounts due to integer division. Typically, `fees` are `rounded up` to ensure the protocol collects at least the intended `minimum fee`. This applies to both the `referrer fee` and `protocol fee` calculations.

Recommendations

Use a `round-up` method for `fee calculation` to ensure accurate `fee collection`.

Monorail: Resolved with @f0667c1d9c...

Zenith: Verified.

[I-4] There is a missing parameter documentation in function comments

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

In `_swapUniswapV2`, the NatSpec comment is missing a description for the `minAmountOut` parameter.

- MonorailAggregator.sol#L1021

```
/**
    * @dev Executes a swap on a Uniswap V2 compatible router.
    * @param router The address of the Uniswap V2 router.
    * @param tokenIn The input token address.
    * @param amountIn The amount of input tokens.
    * @param tokenOut The output token address.
    * @param deadline The transaction deadline.
    */
function _swapUniswapV2(
    address router,
    address tokenIn,
    uint256 amountIn,
    address tokenOut,
@>  uint256 minAmountOut,
    uint256 deadline
) private {
    address[] memory path = new address[](2);
    path[0] = tokenIn;
    path[1] = tokenOut;
```

The `minAmountOut` is an important parameter representing the minimum acceptable output of the `swap` to prevent `slippage losses`. Several other `swap` functions in the codebase are also missing same NatSpec descriptions for this parameter.

Recommendations

Update the NatSpec to include `minAmountOut` and ensure consistency across all `swap` functions.

Monorail: Resolved with @09b07ac6e7...

Zenith: Verified.

[I-5] There is a minor typo and case inconsistency in the comments of the `_swapCloberOrderbook` function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

In `_swapCloberOrderbook`, there are minor issues in the inline comments.

- MonorailAggregator.sol#L1275

```
function _swapCloberOrderbook(
    address router,
    address tokenIn,
    uint256 amountIn,
    address tokenOut,
    uint256 minAmountOut,
    uint256 deadline,
    bytes memory extraParams
) private {
@>  // BookID the the orderbook we are interacting with
    // These parameters are passed as ABI-encoded bytes in extraParams.
    uint192 bookId = abi.decode(extraParams, (uint192));
    if (bookId == 0) revert InvalidRouting();

    ICloberRouter tradeRouter = ICloberRouter(router);

    ICloberRouter.SpendOrderParams[]
        memory params = new ICloberRouter.SpendOrderParams[](1);
    params[0] = ICloberRouter.SpendOrderParams({
        id: ICloberRouter.BookId.wrap(bookId),
        limitPrice: 0,
        baseAmount: amountIn,
        minQuoteAmount: minAmountOut,
@>      // Hookdata is zeroed in this context
        hookData: new bytes(32)
    });
```

1. The word **the** is duplicated in the first comment.
2. **Hookdata** does not match the actual variable name `hookData`, causing slight confusion for readers.

These issues are minor and do not affect functionality, but they reduce code readability and clarity.

Recommendations

Update the comments for clarity and accuracy.

Monorail: Resolved with @fa288f4b1....

Zenith: Verified.

[I-6] There is a documentation mismatch for aggregatePermit2 token input

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

The function `aggregatePermit2` enforces `ERC20`-only inputs through `_validateERC20OnlyInput` and native tokens cannot be used as `tokenIn`.

- MonorailAggregator.sol#L429

```
/**
    * @notice Executes a sequence of trades using Permit2 for ERC20 token
    approval.
    * @dev The owner of the tokens signs a Permit2 message, which allows this
    function

@>  * @param tokenIn The address of the initial input token. Use
    `NATIVE_TOKEN` if sending native currency.
    */
function aggregatePermit2(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 minAmountOut,
    address destination,
    uint256 deadline,
    uint64 referrer,
    uint64 quote,
    Trade[] calldata trades,
    IPermit2.PermitSingle calldata permit,
    bytes calldata signature
) external nonReentrant {
    // Permit2 only handles ERC20 tokens
@>  _validateERC20OnlyInput(tokenIn, tokenOut, amountIn);
```

However, the NatSpec documentation suggests otherwise.

```
* @param tokenIn The address of the initial input token. Use `NATIVE_TOKEN`
    if sending native currency.
```

This creates a misleading inconsistency between the code and its documentation.

Recommendations:

Update the NatSpec documentation to accurately reflect the implementation, clarifying that `tokenIn` must always be an `ERC20 token` when using `aggregatePermit2` and `aggregatePermitted` functions.

Monorail: Resolved with @d1839b725....

Zenith: Verified.

[I-7] Inconsistent error handling between custom errors and string reverts

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

The contract uses **custom errors** in some functions, such as `_wrapper`:

- MonorailAggregator.sol#L1005

```
function _wrapper(
    address router,
    address tokenIn,
    uint256 amountIn,
    address tokenOut
) private {
    if (tokenIn == NATIVE_TOKEN) {
        // Wrap native
        IWrapper(router).deposit{value:  amountIn}();
    } else if (tokenOut == NATIVE_TOKEN) {
        // Unwrap wrapped
        // This router needs to be on the approved native senders list
        // to allow the transfer of native tokens back to us
@>      require(allowedNativeSenders[router], UnauthorizedNativeTransfer());
        IWrapper(router).withdraw(amountIn);
    }
    // Ensure that the amount we wrapped/unwrapped is the same as the
    amountIn just in tokenOut
    uint256 tokenOutBalance = _getBalance(tokenOut);
    require(tokenOutBalance >= amountIn, SlippageExceeded());
}
```

But in `_swapKuruOrderbook`, it uses **string-based errors** instead:

- MonorailAggregator.sol#L1180

```
function _swapKuruOrderbook(
    address router,
    address tokenIn,
    uint256 amountIn,
    address tokenOut,
    uint256 minAmountOut,
    bytes memory extraParams
) private {
    // Validation for Kuru parameters.
@>  require(markets.length > 0, "Invalid markets");
    require(markets.length == isBuy.length, "Markets length mismatch");
    require(
        markets.length == isNativeSend.length,
        "Markets length mismatch"
    );
```

This creates inconsistency in `error handling` across the codebase.

Recommendations:

For maintainability and readability, all revert conditions should consistently use **custom errors** instead of mixing with string-based reverts.

Monorail: Fixed in the commit @4eafe190b....

Zenith: Verified.

[I-8] The `_calculateAmounts` function contains a redundant condition check

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

Inside `_calculateAmounts`, the code checks `if (feeInfo.totalFeeBps > 0)` twice: once in the outer `if` statement and again immediately inside it.

- MonorailAggregator.sol#L842

```
function _calculateAmounts(
    address tokenOut,
    uint64 referrer
)
    private
    view
    returns (
        uint256 destinationAmount,
        FeeDetails memory referrerDetails,
        uint256 protocolFeeAmount
    )
{
    // A valid, active referrer is provided
    if (feeInfo.receiver ≠ address(0) && feeInfo.totalFeeBps > 0) {
@>      if (feeInfo.totalFeeBps > 0) {
            uint256 totalFeeAmount = (finalAmountOut *
                feeInfo.totalFeeBps) / BASIS_POINTS;

            // Calculate the referrer's share of the total fee
            uint256 refAmount = (totalFeeAmount *
                feeInfo.referrerShareBps) / BASIS_POINTS;
            referrerDetails = FeeDetails(refAmount, feeInfo.receiver);

            // The rest goes to the protocol
```

```
                protocolFeeAmount = totalFeeAmount - refAmount;
                destinationAmount = finalAmountOut - totalFeeAmount;
                return (destinationAmount, referrerDetails, protocolFeeAmount);
            }
        }
        ...
}
```

The inner condition is redundant because the outer condition already enforces `feeInfo.totalFeeBps > 0`.

Recommendations:

```
function _calculateAmounts(
    address tokenOut,
    uint64 referrer
)
    private
    view
    returns (
        uint256 destinationAmount,
        FeeDetails memory referrerDetails,
        uint256 protocolFeeAmount
    )
{
    // A valid, active referrer is provided
    if (feeInfo.receiver ≠ address(0) && feeInfo.totalFeeBps > 0) {
        if (feeInfo.totalFeeBps > 0) {
            ...
        }
    }
    ...
}
```

Monorail: Resolved with @c81b6d944....

Zenith: Verified.

[I-9] The initialization of defaultProtocolFeeBps is unnecessary

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

In the `initialize` function, the contract sets `defaultProtocolFeeBps = 0`.

- MonorailAggregator.sol#L291

```
function initialize(
    address initialOwner,
    address permit2Address
) external initializer {
    __Ownable_init(initialOwner);
    __ReentrancyGuard_init();
    __UUPSUpgradeable_init();

    if (permit2Address == address(0)) revert Permit2Disabled();
    permit2 = IAllowanceTransfer(permit2Address);
@>  defaultProtocolFeeBps = 0;
}
```

In Solidity, state variables of type `uint` are automatically initialized to `0` by default. Explicitly setting `defaultProtocolFeeBps = 0` during initialization has no effect.

Recommendations:

```
function initialize(
    address initialOwner,
    address permit2Address
) external initializer {
    __Ownable_init(initialOwner);
    __ReentrancyGuard_init();
    __UUPSUpgradeable_init();

    if (permit2Address == address(0)) revert Permit2Disabled();
    permit2 = IAllowanceTransfer(permit2Address);
   defaultProtocolFeeBps = 0;
}
```

Monorail: Resolved with @300e29ecf....

Zenith: Verified.

[I-10] Some validations are not implemented in the `setReferrerFeeBps` function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol

Description:

The `setReferrerFeeBps` function allows the `contract owner` to configure `referral fee` settings.

- MonorailAggregator.sol#L367

```
function setReferrerFeeBps(
    uint64 referrer,
    address receiver,
    uint16 totalFeeBps,
    uint16 referrerFeeBps
) external onlyOwner {
    if (totalFeeBps < MIN_FEE_BPS || totalFeeBps > MAX_FEE_BPS) {
        revert InvalidAmount();
    }

    // Update the referrer fee for the caller
    // Setting totalFeeBps to 0 disables the referrer fee
    ReferrerFeeInfo memory feeInfo;
    feeInfo.receiver = receiver;
    feeInfo.totalFeeBps = totalFeeBps;
    feeInfo.referrerShareBps = referrerFeeBps;

    referrerFees[referrer] = feeInfo;

    emit ReferrerFeeUpdated(
        referrer,
        feeInfo.receiver,
        feeInfo.totalFeeBps,
        feeInfo.referrerShareBps
    );
}
```

There are some missing validations.

1. `receiver` validation is missing:
   - The function allows `receiver = address(0)`.
   - As a result, the transaction would be reverted.
2. `referrerFeeBps` validation is missing
   - There is no check for `referrerFeeBps`.
   - If `referrerFeeBps` is configured higher than `100%`, any `trade` using this `referrer setting` will revert during `fee distribution logic`.

Recommendations:

```
function setReferrerFeeBps(
    uint64 referrer,
    address receiver,
    uint16 totalFeeBps,
    uint16 referrerFeeBps
) external onlyOwner {
    if (totalFeeBps < MIN_FEE_BPS || totalFeeBps > MAX_FEE_BPS) {
        revert InvalidAmount();
    }
+^^Iif (referrerFeeBps > BASIS_POINTS) {
+       revert InvalidAmount();
+   }
+   if (receiver == address(0)) revert InvalidAddress();
    ...
}
```

Monorail: Resolved with @82577f2ce....

Zenith: Verified.

[I-11] Incorrect comment

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MonorailAggregator.sol#L81

Description:

In `MonorailAggregator`, we define the `MAX_WEIGHT` constant with a descriptive comment:

```
/// @notice The maximum allowed weight for a single leg in basis points
    (100000000 = 100%).
uint32 private constant MAX_WEIGHT = 100000000;
```

In the comment, we describe the value as being denominated in basis points, but it is actually not denominated in basis points, potentially causing confusion.

Recommendations:

Adjust the comment to not indicate that the value is denominated in basis points:

```
/// @notice The maximum allowed weight for a single leg in basis points (
    100000000 = 100%).
/// @notice The maximum allowed weight for a single leg (100000000 = 100%).
uint32 private constant MAX_WEIGHT = 100000000;
```

Monorail: Resolved with @55846b4....

Zenith: Verified.

[I-12] Missing NatSpec comment

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- MonorailAggregator.sol#L618

Description:

In `MonorailAggregator._executeCore`, we include the `quote` parameter. However, we don't have a `NatSpec` comment describing that parameter as we do with the other parameters.

Recommendations:

Add a NatSpec comment describing the `quote` parameter.

Monorail: Resolved with @b13db3d....

Zenith: Verified.

[I-13] Trade weight logic may lead to unexpected behavior

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- MonorailAggregator.sol#L652-L654

Description:

In `MonorailAggregator._executeCore`, each `trade` has a `weight` value which is used to determine the relative amount of a given token to be used as input for that trade:

```
uint256 innerAmountIn = (tokenInBalance * trade.weight) /
    MAX_WEIGHT;
tokenInBalance -= innerAmountIn;
```

Since we decrement the `tokenInBalance` each time, it's important to recognize that the weight distribution will not simply be the relative weight of each trade, but instead must be adjusted according to ordering and remaining `tokenInBalance`. Consider the following example where we assume that the trade weights are relative distribution amounts:

- We execute two trades with the same input token, each with a `weight` of 50% (5000 bps), with an `amountIn` of 1000
- The first trade computes `innerAmountIn` as `1000 * 5000 / 10000 = 500`
- `tokenInBalance is decremented by` innerAmountIn: 1000 - 500 = 500‘
- The second trade computes `innerAmountIn` as `500 * 5000 / 10000 = 250`

As we can see from the example, while we expected each trade to be 50% of the total `amountIn`, the second trade only consumes 50% of the remaining amount.

Note that this appears to be intended behavior based on testing logic implemented, see: `MonorailRouterTests.testWeightedSwapDistribution`. However, this may likely be unexpected behavior for users and can be made more user friendly as recommended below.

Recommendations:

Don't decrement the `tokenInBalance` after computing `innerAmountIn`:

```
uint256 innerAmountIn = (tokenInBalance * trade.weight) /
    MAX_WEIGHT;
tokenInBalance -= innerAmountIn;
```

This will allow for an even trade weight distribution.

Monorail: Acknowledged. This is intended behavior.