yAudit

February 2026

Prepared for
Vfat

Audited by
HHK
adriro

Vfat - MultiSwapRouter

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

The MultiSwapRouter is a unified swapping router that provides a single interface for executing multi-hop and split swaps across numerous DEX types, including Uniswap V2/V3/V4, Solidly/Velodrome/Aerodrome, Algebra, Camelot, Slipstream, and Blackhole.

1.2 Audit Scope

This audit covers two smart contracts totaling approximately 1200 lines of code across 7 days of review.

```
contracts/
├── MultiSwapRouter.sol
└── security
    └── RouterAllowlist.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	5
Informational	11

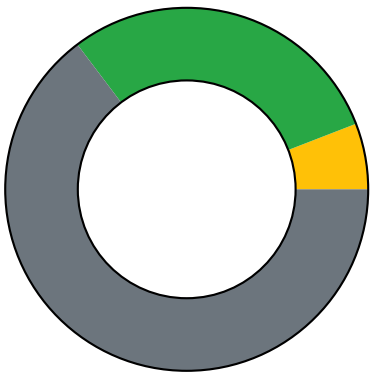

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The MultiSwapRouter is a well-structured codebase with solid test coverage and good documentation. The main concerns identified relate to the router's handling of edge cases across its numerous DEX integrations.

2 Audit Overview

2.1 Project Information

Protocol Name: Vfat
Repository: https://github.com/vfat-io/sickle-contracts
Commit URLs:

- `9b0c9daa347b4bb47a5dcaa9e9269f1dad4699b7`
- `db1bcc93f630f79a58aeb68e47867c27e4a9f284`

2.2 Audit Team

HHK, adriro

2.3 Audit Timeline

The audit was conducted from February 13 to 23, 2026.

2.4 Audit Resources

Code repositories and documentation

Category	Mark	Description
Access Control	Good	Admin-accessible functions have proper access control.
Mathematics	Good	Mathematical operations are correctly implemented despite some unsafe casts.
Complexity	Average	The numerous DEX integrations introduce inherent complexity. An issue was identified where user funds could become stuck in the router under certain edge cases.
Libraries	Good	The implementation uses the Solmate library.
Decentralization	Good	The protocol enables permissionless swapping. The admin maintains a router/factory allowlist and controls fee settings.
Code Stability	Good	The codebase remained stable during the review.
Documentation	Good	The contracts are well-documented with clear comments and proper NatSpec.
Monitoring	Good	Events for state-changing functions are in place.
Testing and verification	Good	The test suite provides solid functional coverage with thorough unit tests and broad fork-based integration tests for the router.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

2.7.1 Direct pool swaps strand unused input on partial fills

Users lose funds when swaps partially fill, as the router pre-pulls the full input but only the consumed portion is used. The remainder is stranded — either in the router (recoverable by admin only) or in an external router (unrecoverable entirely).

Technical Details

The router's `_pullInput()` unconditionally transfers the full `amountIn` from the caller into the router. Multiple swap paths are then vulnerable to partial fills where the pool consumes less than the full input:

1. Direct pool swaps (UniswapV3Pool / AlgebraPool). `_swapAlgebraPool()` and `_swapUniswapV3Pool()` call `pool.swap()` with `amountSpecified = int256(amountIn)`. If liquidity is insufficient, the pool partially fills. The callbacks (`uniswapV3SwapCallback()`, `algebraSwapCallback()`) pay only the actual positive delta, not the full amount. The difference remains stranded in the router, recoverable only via admin `sweep()` to the fee collector.

2. UniswapV4. `_swapUniswapV4()` builds a `SWAP_EXACT_IN_SINGLE + SETTLE_ALL + TAKE_ALL` action sequence. `SETTLE_ALL` settles based on the actual swap delta, not the full `amountIn`. On partial fills, Permit2 pulls only the consumed amount from the router. The unconsumed tokens remain stranded in the router after the Permit2 approval is revoked at L622-624.

3. VelodromeUniversalRouter (CL route). `_swapVelodromeUniversalRouter()` pushes the full `amountIn` to the Velodrome router via `safeTransfer` before executing the swap. When using the CL path (`command = 0x00`), the underlying Slipstream pool can partially fill like any V3-style pool. The unconsumed tokens remain stranded **on the Velodrome router itself**, not on the MultiSwapRouter. The MSR admin has no ability to recover these tokens — they are permanently lost to the user unless the Velodrome router has its own sweep mechanism controlled by a separate admin.

Impact

Medium. Users permanently lose unconsumed input on partial fills. For direct pool and V4 swaps, stranded tokens remain in the router and are swept to the fee collector. For VelodromeUniversalRouter CL swaps, stranded tokens are sent to the external router and are entirely unrecoverable by the MSR admin.

Recommendation

For direct pool swaps and V4, revert when the consumed amount differs from `amountIn` to prevent partial fills entirely or compute the delta and transfer back any unused tokens.
For VelodromeUniversalRouter, switch from the token push pattern to approve-call-revoke with `payerIsUser = true`, consistent with other router-based swaps. This keeps unconsumed tokens in the MSR where they can at least be swept.

Developer Response

Fixed in PR#653 & PR659.

2.8 Low Findings

2.8.1 Slippage check in `_swapAndCollectFee()` does not account for fees

In `_swapAndCollectFee()`, the `minSwapOut` slippage check is performed on the raw swap output before protocol fees are deducted. This is inconsistent with the regular swap path in `_applyFeeAndTransfer()`, where slippage is checked after fees, and can cause users to receive less than their intended minimum.

Technical Details

In the swap-and-bridge flow, `_swapAndCollectFee()` validates the swap output against `minSwapOut` at line `if (currentAmount < minSwapOut)`, then subsequently deducts fees from the output amount. This means a user setting `minSwapOut` to protect against slippage may still receive `minSwapOut - fee` worth of tokens forwarded to the bridge.
In contrast, the standard swap path uses `_applyFeeAndTransfer()`, which first calculates `amountAfterFee = totalOutput - fee` and then checks `if (amountAfterFee < minAmountOut)`. This correctly ensures the user receives at least `minAmountOut` after all deductions.
The discrepancy means that in the bridge path, the effective minimum output is lower than what the user specified by up to `feeBps / BPS_DENOMINATOR` of the output amount (currently up to 1%).

Impact

Low. Users bridging through the swap-and-bridge flow may receive less than their intended minimum output amount. The difference is bounded by the protocol fee (up to 1%), but the slippage protection does not function as expected.

Recommendation

Move the `minSwapOut` check to after the fee is deducted, or adjust the check to account for the fee, consistent with how `_applyFeeAndTransfer()` handles it in the regular swap path.

Developer Response

Fixed in `05c175fd`.

2.8.2 Bridge calls pass user-controlled calldata without function selector validation

`_executeBridgeAndRefund()` calls the allowlisted bridge contract with fully user-supplied calldata, allowing any function on that contract to be invoked. Unlike swap paths where the MSR constructs calldata internally, bridge callers control the function selector entirely.

Technical Details

In `_executeBridgeAndRefund()`, the bridge call is:

```
(bool success, bytes memory result) =
    bridgeContract.call{ value: bridgeNativeValue }(bridgeCalldata);
```

The `bridgeCalldata` is passed through verbatim from the external caller. While `bridgeContract` is validated against the router allowlist via `allowlist.requireAllowed(bridgeContract)`, no restriction is placed on which function is called.
Bridge contracts typically expose multiple entry points beyond their bridging function — admin functions, token recovery, alternative transfer modes, or proxy upgrade functions. An allowlisted bridge contract with a secondary function that transfers tokens to a caller-specified address (rather than bridging cross-chain) could be misused.

Impact

Low. Requires the allowlisted bridge contract to have a secondary function exploitable within the approve-call-revoke flow. The consumption check limits the blast radius to the current call's deposited amount. However, the attack surface is broader than necessary.

Recommendation

Add a function selector allowlist to the `RouterAllowlist` contract, mapping `(bridgeContract, bytes4 selector) → bool`. Validate the first 4 bytes of `bridgeCalldata` against this allowlist before executing the call:

```
bytes4 selector = bytes4(bridgeCalldata[:4]);
allowlist.requireAllowedSelector(bridgeContract, selector);
```

With a selector allowlist in place, the `nonReentrant` guard on `swapAndBridge()` and `bridge()` becomes likely unnecessary — the bridge contract is allowlisted and can only execute pre-approved functions, matching the trust model of `swap()` and `swapSplit()` which operate under the same allowlist without a reentrancy guard.

Developer Response

Fixed in 05c175fd.

2.8.3 `_wrapWETH` can consume pre-existing ETH held by the router

`_wrapWETH()` validates the wrap amount against `address(this).balance`, the router's entire ETH balance, without distinguishing between ETH provided in the current call and pre-existing ETH. Combined with the lack of step token continuity, an attacker can wrap and extract ETH that was not supplied by their transaction.

Technical Details

`_wrapWETH()` checks `address(this).balance < amountIn` but does not track how much ETH belongs to the current operation versus pre-existing balances (from accidental transfers, `selfdestruct` forced sends, or refunds). Since `WrapWETH` and `UnwrapWETH` steps are exempt from the router allowlist check in `_executeStep()`, and no per-step token continuity is enforced, an attacker can:

1. Execute a swap through a permissionless pool on an allowlisted factory that returns an attacker-chosen `currentAmount`.
2. Use that `currentAmount` as input to a `WrapWETH` step, which wraps that many wei from the router's global ETH balance.
3. Receive the resulting WETH (or unwrap it back to ETH) through `_applyFeeAndTransfer()`.

This bypasses the admin-only `sweepETH()` path intended to recover stranded ETH.

Impact

Low. Any ETH accumulated on the router can be extracted by any caller. Loss is bounded to the router's pre-existing ETH balance and does not directly drain user funds.

Recommendation

Introduce per-call ETH accounting to isolate ETH from the current operation. Record the initial balance at the start of `swap()` / `swapSplit()` and pass it through to `_wrapWETH()`:

```
uint256 initialEthBalance = address(this).balance - msg.value;
```

Then in `_wrapWETH()`, replace the current balance check:

```
uint256 callEthBalance = address(this).balance - initialEthBalance;
if (callEthBalance < amountIn) {
    revert InsufficientEthBalance();
}
```

Additionally, enforcing step token continuity (`steps[i].tokenIn == steps[i-1].tokenOut`) as recommended in other findings would prevent arbitrary `WrapWETH` insertion into multi-step sequences.

Developer Response

Token continuity checks (PR #654) and `amountIn == msg.value` validation in `_pullInput` already block the attack path to `_wrapWETH`.

2.8.4 Missing `tokenOut` validation and step token continuity allow draining router-held balances

The router lacks two related validation checks that together allow an attacker to extract any ERC20 or ETH balance held by the router. First, `_swapUniswapV3Pool()` validates only `tokenIn` against the pool but not `tokenOut`. Second, neither `_swap()` nor `_executeSingleRoute()` enforce that each step's `tokenIn` matches the previous step's `tokenOut`. These can be exploited independently or combined for more complex drains.

Technical Details

1. V3 direct pool `tokenOut` spoofing. `_validateV3PoolAndDirection()` confirms `tokenIn` is either `token0` or `token1` but ignores `step.tokenOut`. The swap at `_swapUniswapV3Pool()` computes `amountOut` from pool deltas without reference to `step.tokenOut`. An attacker can reference a legitimate A/B pool, set `tokenIn = A` and `tokenOut = X` (an unrelated token held by the router), and `_applyFeeAndTransfer()` will transfer `amountOut` units of X to the attacker while the actual pool output B remains stranded. Note that `_swapAlgebraPool()` already validates both tokens via `_validateAlgebraPoolAndDirection()`, making only the V3 path vulnerable.
2. Step token discontinuity. `_swap()` pulls only the first step's input from the caller and threads a numeric `currentAmount` through subsequent steps without verifying `steps[i].tokenIn == steps[i-1].tokenOut`. Similarly, `_executeSingleRoute()` in `swapSplit()` applies no per-step continuity check. An attacker can construct a multi-step swap where step 0 swaps a small amount of their own token, then step 1 declares a completely different `tokenIn` corresponding to a valuable token the router holds. The router approves and spends its pre-existing balance of that token via `_callRouterWithApproval()`, transferring the proceeds to the attacker. This works with any dex type, not just V3 direct pools.
3. Combined exploitation. Both vectors can be chained: a V3 pool step with spoofed `tokenOut` manufactures an arbitrary `currentAmount`, then a subsequent step with a discontinuous `tokenIn` spends a different router-held token using that spoofed amount.

Impact

Low. Drains any ERC20 or ETH balance residing on the router (dust, accidental transfers, airdrops, stranded tokens from prior swaps). Does not directly compromise user funds passing through well-formed swaps. Requires the router to hold a non-trivial balance of the target token.

Recommendation

Validate `tokenOut` against the pool's opposite token in `_validateV3PoolAndDirection()`:

```
 function _validateV3PoolAndDirection(
     address pool,
-    address tokenIn
+    address tokenIn,
+    address tokenOut
 ) internal view returns (bool zeroForOne) {
     ...
```

```
        if (tokenIn == token0) {
+           if (tokenOut != token1) revert InvalidTokenIn();
            return true;
        }
        if (tokenIn == token1) {
+           if (tokenOut != token0) revert InvalidTokenIn();
            return false;
        }
        revert InvalidTokenIn();
    }
```

Enforce per-step token continuity in `_swap()`:

```
    function _swap(
        SwapStep[] calldata steps,
        uint256 amountIn,
        uint256 minAmountOut,
        address recipient,
        uint256 deadline
    ) internal returns (uint256 amountOut) {
        if (block.timestamp > deadline) revert Expired();
        if (steps.length == 0) revert InvalidStepsLength();

        _pullInput(steps[0].tokenIn, amountIn);

        uint256 currentAmount = amountIn;
        uint256 minOutBeforeFee = _requiredOutputBeforeFee(minAmountOut);
        for (uint256 i; i < steps.length; i++) {
+           if (i > 0 && steps[i].tokenIn != steps[i - 1].tokenOut) {
+               revert InvalidTokenPath();
+           }
            uint256 stepMinOut;
            if (i == steps.length - 1 && steps[i].dexType == DexType.UniswapV4) {
                stepMinOut = minOutBeforeFee;
            }
            currentAmount = _executeStep(steps[i], currentAmount, stepMinOut);
        }
        ...
    }
```

Apply the same continuity check in `_executeSingleRoute()` for `swapSplit()`:

```
    function _executeSingleRoute(
        SwapStep[] calldata steps,
        uint256 amountIn
    ) internal returns (uint256 currentAmount) {
        currentAmount = amountIn;
        for (uint256 s; s < steps.length; s++) {
+           if (s > 0 && steps[s].tokenIn != steps[s - 1].tokenOut) {
+               revert InvalidTokenPath();
+           }
            currentAmount = _executeStep(steps[s], currentAmount, 0);
        }
    }
```

Developer Response

Fixed in PR#654.

2.8.5 `_requiredOutputBeforeFee()` ceiling formula causes avoidable V4 reverts

The pre-fee minimum for UniswapV4 final steps uses a ceiling division that doesn't match the floor-rounded fee deduction, rejecting valid swaps at the V4 `TAKE_ALL` level.

Technical Details

`_requiredOutputBeforeFee()` inverts the **continuous** fee formula via ceiling division, but `_applyFeeAndTransfer()` uses **floor division** for the actual fee:

```
// MultiSwapRouter.sol#L849 — ceiling of continuous inverse
return (minAmountOut * BPS_DENOMINATOR + denom - 1) / denom;

// MultiSwapRouter.sol#L299 — actual fee uses floor
fee = (totalOutput * feeBps) / BPS_DENOMINATOR;
```

Because the fee rounds down, the user retains more than the continuous formula predicts, so the ceiling-based inverse overshoots the true minimum by 1 unit.
Additionally, `swapSplit()` never applies this V4 pre-fee minimum — `_executeSingleRoute()` always passes `0` as `stepMinOut`, so the same V4 swap that reverts via `swap()` succeeds via `swapSplit()` with a single route.

Impact

Low. Swaps where the V4 output falls exactly at the 1-unit boundary between the true minimum and the ceiling-derived bound will revert unnecessarily via `swap()`, wasting gas. The inconsistency between `swap()` and `swapSplit()` complicates integrator routing logic. No funds are at risk.

Recommendation

Replace the ceiling formula with the correct discrete inverse that accounts for floor-rounded fee deduction, and apply the same logic in `swapSplit()` for consistency.
Alternatively consider passing `0` like in `swapSplit()`.

Developer Response

Fixed in commit 345d0b9c.

2.9 Gas Savings Findings

2.9.1 Redundant `balanceOf()` call in `_executeBridgeAndRefund()`

Technical Details

`_executeBridgeAndRefund()` queries the `bridgeToken` balance twice, the first while computing the `available` amount (line 480) and again to assign `balanceBeforeBridge` (line 491).

```
478:         // Cap bridgeAmount to what this call actually deposited (post-fee)
479:         uint256 available =
480:             IERC20(bridgeToken).balanceOf(address(this))
481:                 - initialBridgeTokenBalance;
482:         if (bridgeAmount > available) {
483:             revert BridgeAmountExceedsAvailable(bridgeAmount, available);
484:         }
485:
486:         // Approve bridge contract for exactly bridgeAmount
487:         _approve(bridgeToken, bridgeContract, bridgeAmount);
488:
489:         // Record balance before bridge call for consumption check
490:         uint256 balanceBeforeBridge =
491:             IERC20(bridgeToken).balanceOf(address(this));
```

Impact

Gas.

Recommendation

Query the balance once and use this value in both locations.

Developer Response

Fixed in `05c175fd`.

2.9.2 `_preCallEthBalance` uses storage instead of a function parameter

Technical Details

Both `swapAndBridge()` and `bridge()` compute `_preCallEthBalance = address(this).balance - msg.value` and store it in a `uint256 private` storage variable. The only consumer is `_executeBridgeAndRefund()`, which reads it once for the ETH refund calculation. This costs a cold SSTORE (~5k gas) on write and a warm SLOAD (~100 gas) on read per bridge call, versus near-zero cost for passing a stack variable through the call chain. The value also persists in storage after the function returns, which is unnecessary.

Impact

Informational. ~5k gas overhead per bridge call with no functional benefit over a function parameter.

Recommendation

Remove the `_preCallEthBalance` storage variable. Compute the value locally in each entry point and pass it as a parameter to `_executeBridgeAndRefund`.

Developer Response

Fixed in 05c175fd.

2.10 Informational Findings

2.10.1 `bridge()` has no deadline check

Technical Details

`swapAndBridge()` accepts a `deadline` parameter and reverts if `block.timestamp > deadline`, but `bridge()` has no deadline mechanism. Transactions can sit in the mempool and execute at an arbitrary future time.

Impact

Informational. Users cannot protect against stale bridge transactions executing after conditions have changed.

Recommendation

Add a `deadline` parameter to `bridge()` consistent with `swapAndBridge()`.

Developer Response

Fixed in 05c175fd.

2.10.2 `bridgeAmount` parameter and redundant check in `_executeBridgeAndRefund`

`_executeBridgeAndRefund()` takes a user-supplied `bridgeAmount`, checks it against `available`, then approves exactly that amount. This check is unnecessary — the approval itself caps what the bridge can pull, and the existing consumption check and refund logic handle all cases.

Technical Details

In `_executeBridgeAndRefund()`, the flow is:

```
uint256 available = IERC20(bridgeToken).balanceOf(address(this)) -
initialBridgeTokenBalance;
if (bridgeAmount > available) {
    revert BridgeAmountExceedsAvailable(bridgeAmount, available);
}
_approve(bridgeToken, bridgeContract, bridgeAmount);
```

Both `swapAndBridge()` and `bridge()` already know the effective amount available after pulling input and collecting fees. Each entry point should compute this amount and pass it directly to `_executeBridgeAndRefund`, removing the `bridgeAmount` parameter from both the internal function and the external interface.

Impact

Informational. Simplifies the external interface, removes a redundant parameter and check, and makes the approval inherently safe by construction (always set to exactly what the current call deposited post-fee).

Recommendation

Compute the effective post-fee amount in each entry point (`swapAndBridge` from `_swapAndCollectFee`'s output, `bridge` from `amountReceived - fee`) and pass it to `_executeBridgeAndRefund` as the approval amount. Remove the `bridgeAmount` parameter from both the external functions and the internal helper, along with the `BridgeAmountExceedsAvailable` check and update `BridgeDidNotConsumeTokens` check.

Developer Response

Fixed in 05c175fd.

2.10.3 Multiple gas and code quality improvements

Several minor inefficiencies and code duplication patterns were identified across both contracts.

Technical Details

1. Redundant first iteration in `_validateAndSumAmounts()`. `_validateAndSumAmounts()` derives `tokenIn` and `tokenOut` from `routes[0]` at L242-L243, then loops from `i = 0` and checks every route against those values. The `i = 0` iteration compares `routes[0]` against itself — always true. The loop could start at `i = 1`.
2. Unused `amountIn` in Algebra callback data. `_swapAlgebraPool()` encodes `amountIn` in the callback data, but `algebraSwapCallback()` skips it during decoding. The callback pays `amountToPay` from the pool's reported delta, not the encoded value. The encoding and decoding of this unused field wastes gas.
3. Wrap/unwrap uses `step.router` instead of `wrappedNative` directly. `_wrapWETH()` and `_unwrapWETH()` validate that `step.router == wrappedNative`, then call

`IWETH(step.router)`. Using `IWETH(wrappedNative)` directly would eliminate the `step.router != wrappedNative` check and simplify both functions.

4. Duplicated swap-and-pop pattern in `RouterAllowlist`. `_setRouter()`, `_setFactory()`, and `_setHook()` each implement identical swap-and-pop array management logic. This could be consolidated into a single reusable internal function or replaced with OpenZeppelin's `EnumerableSet`.

Impact

Informational. Minor gas overhead and code maintainability concerns. No security impact.

Recommendation

Implement the suggested changes.

Developer Response

Partially fixed in PR#658. Swap-and-pop pattern in `RouterAllowlist` kept as is.

2.10.4 Different `setAdmin()` zero-address check between Router and Allowlist

`Admin.setAdmin()` allows setting the admin to `address(0)`, which would permanently brick all allowlist management. While `MultiSwapRouter.setAdmin()` guards against this.

Technical Details

```
// Admin.sol#L46-L51 — no zero-address check
function setAdmin(address newAdmin) external onlyAdmin {
    emit AdminSet(admin, newAdmin);
    admin = newAdmin;
}

// MultiSwapRouter.sol#L344-L349 — has zero-address check
function setAdmin(address newAdmin) external {
    if (msg.sender != admin) revert NotAdmin();
    if (newAdmin == address(0)) revert InvalidAddress();
    ...
}
```

Impact

Informational. An admin mistake could permanently lock out `RouterAllowlist` management. No routers, factories, or hooks could ever be added or removed.

Recommendation

Add a zero-address check to `Admin.setAdmin()` or override it in `RouterAllowlist` to match `MultiSwapRouter`.

Developer Response

Fixed in PR#656.

2.10.5 Unnecessary `_tryComputeAlgebraPoolAddress()` call in `_isAlgebraPoolFromFactory()`

In `_isAlgebraPoolFromFactory()`, after checking `poolByPair`, the function calls `_tryComputeAlgebraPoolAddress()` to compute the deterministic pool address. This is unnecessary because Algebra pools are either registered in the `poolByPair` mapping or in the `customPoolByPair` mapping.

Technical Details

The function first queries `poolByPair` on the factory. If that fails or returns a different address, it falls through to `_tryComputeAlgebraPoolAddress()`, which computes a deterministic `CREATE2` address. Matching a `CREATE2` address does not prove the pool exists. Since all legitimate pools should be registered in either `poolByPair` or `customPoolByPair`, this computed address check adds no security value and could potentially validate a non-existent pool.

Impact

Informational.

Recommendation

Consider removing the `_tryComputeAlgebraPoolAddress()` check. If a pool is not found in `poolByPair`, skip directly to the `customPoolByPair` check when a `customDeployer` is provided, and return `false` otherwise.

Developer Response

Fixed in commit `f9ef2c04`. Removed redundant computePoolAddress check in Algebra pool validation.

2.10.6 Natspec for `_swapAlgebra()` incorrectly states `dexData` should be empty for classic Algebra

The natspec on `_swapAlgebra()` states that for classic Algebra (QuickSwap V3), `dexData` should be empty. However, the code only checks `step.dexData.length == 32` for the Algebra Integral branch and falls through to the classic branch for any other length, including non-empty values.

Technical Details

The function branches on `step.dexData.length == 32` to distinguish Algebra Integral from classic Algebra. Any `dexData` that is not exactly 32 bytes (including non-empty values of other lengths) will execute the classic Algebra path. The natspec stating `dexData should be empty` is misleading since the code does not enforce that constraint.

Impact

Informational.

Recommendation

Update the natspec or enforce `dexData` to be empty.

Developer Response

Fixed in commit `fd5136d4`. Reflect actual behavior (any dexData length != 32 takes the classic path).

2.10.7 Missing `address(0)` check for `AlgebraPool` and `UniswapV3Pool` in `_executeStep()`

In `_executeStep()`, the `tokenIn == address(0)` check is skipped for `AlgebraPool` and `UniswapV3Pool` dex types, even though these pool-level swap functions do not support native ETH.

Technical Details

The check that reverts when `tokenIn == address(0)` is excluded for `UniswapV3Pool` and `AlgebraPool`. These direct pool swap functions (`IUniswapV3Pool.swap()` and their Algebra equivalents) operate on ERC20 tokens exclusively and do not handle native ETH. Passing `address(0)` as `tokenIn` for these dex types would result in a failed swap or unexpected behavior downstream. Including them in the `address(0)` check would provide a clearer revert.

Impact

Informational.

Recommendation

Remove `AlgebraPool` and `UniswapV3Pool` from the exclusion list in the `tokenIn == address(0)` check, or add separate validation within their respective swap functions.

Developer Response

Fixed in PR#656.

2.10.8 Fee rounding to zero on small swaps enables fee avoidance

Technical Details

In `_applyFeeAndTransfer()`, the fee is computed as `fee = (totalOutput * feeBps) / BPS_DENOMINATOR`. When `totalOutput * feeBps < 10_000`, integer division truncates the fee to zero. Traders can split volume across multiple small calls that each stay below this threshold to avoid paying protocol fees entirely.

Impact

Informational. Protocol fee under-collection relative to the configured `feeBps`. Most relevant for low-decimal or high-unit-value output tokens.

Recommendation

Enforce a minimum fee of 1 unit when `feeBps > 0` and `totalOutput > 0` to eliminate the zero-fee region.

Developer Response

Acknowledged — skipped as negligible (fee on a dust-sized swap rounding to zero has no practical impact).

2.10.9 Direct pool swaps derive output from pool deltas instead of actual balance change

`_swapUniswapV3Pool()` and `_swapAlgebraPool()` compute `amountOut` from the pool's returned swap deltas rather than measuring the router's actual `tokenOut` balance increase. If the reported delta overstates what the router received (e.g., fee-on-transfer output tokens), `_applyFeeAndTransfer()` may consume pre-existing `tokenOut` dust held by the router.

Technical Details

Both `_swapUniswapV3Pool()` and `_swapAlgebraPool()` calculate output as:

```
1 uint256 amountOut = zeroForOne ? uint256(-amount1) : uint256(-amount0);
```

This trusts the pool's accounting. Router-based swaps (`_swapViaRouter()`) instead use `_balanceOf()` / `_balanceDelta()` to reconcile actual received amounts. Direct pool swaps bypass this safeguard, so any mismatch between reported and received amounts could cause `_applyFeeAndTransfer()` to pull from pre-existing router balances.

Impact

Informational. Exploitation requires a validated pool with a fee-on-transfer output token and pre-existing `tokenOut` dust on the router. Loss is capped at whatever dust balance exists for that token.

Recommendation

Use balance-before/after measurement consistent with router-based swaps instead of relying on pool deltas.

Developer Response

Fixed in PR#657.

2.10.10 Non-standard token mechanics can cause accounting mismatches that leak router-held balances

The router's accounting assumes standard ERC20 transfer behavior across multiple paths. Fee-on-transfer, rebasing, and reflection tokens can cause mismatches between expected and actual balances, allowing callers to extract pre-existing router-held tokens or causing reverts.

Technical Details

Three distinct paths are affected:
1. Direct pool swaps trust pool deltas for output accounting. `_swapUniswapV3Pool()` and `_swapAlgebraPool()` compute `amountOut` from the pool's returned deltas rather than measuring actual balance change. If the output token charges transfer fees, the router receives less than reported, and `_applyFeeAndTransfer()` consumes pre-existing router balance to cover the shortfall. Router-based swaps avoid this via `_balanceOf()` / `_balanceDelta()`, but direct pool swaps bypass that safeguard.
2. Input pull does not verify actual received amount. `_pullInput()` transfers the full nominal `amountIn` via `safeTransferFrom` without measuring the balance delta. For fee-on-transfer input tokens, the router receives less than `amountIn` but approves and spends the full nominal amount downstream in `_callRouterWithApproval()`, drawing from any pre-existing `tokenIn` balance on the router.
3. Balance-delta output accounting is unsafe for rebasing tokens. `_swapViaRouter()` measures output via `_balanceDelta()`. For rebasing or reflection tokens, balance changes during the external call get misattributed as swap output (positive rebase leaks router holdings) or cause underflow reverts (negative rebase) in `_balanceDelta()`.
In all three cases, exploitation requires the router to hold a pre-existing balance of the affected token and a non-standard token to be used in the swap path.

Impact

Informational. Callers can extract pre-existing router-held balances (dust, accidental transfers, stranded tokens) when non-standard tokens are involved. Does not directly compromise user-deposited funds. Loss is capped at the router's holdings of the affected token.

Recommendation

Either explicitly block non-standard tokens or apply consistent balance-before/after measurement across all paths:

- In `_pullInput()`, measure actual received amount and revert if it differs from `amountIn`.
- In `_swapUniswapV3Pool()` and `_swapAlgebraPool()`, use `_balanceOf()` / `_balanceDelta()` instead of pool deltas, consistent with router-based swaps.
- In `_balanceDelta()`, add an explicit underflow check to produce a clear revert instead of a silent panic.

If non-standard tokens are intentionally unsupported, document this restriction explicitly.

Developer Response

Fixed in PR#657.

2.10.11 Allowlisting a zero address or EOA as router enables silent no-op swaps

`RouterAllowlist._setRouter()` accepts any address without validation, including `address(0)` and EOAs. When a swap step targets these invalid routers, the low-level call in `_callRouter()` succeeds silently (no code executed), producing zero output while input tokens remain stranded in the router.

Technical Details

`_setRouter()` does not validate that the router address is a contract. `_callRouter()` performs `router.call{value: value}(data)` and only reverts if `success == false`. Since low-level calls to `address(0)` or EOAs return `success = true`, the swap step completes as a no-op. Input tokens pulled into the router are only recoverable via admin `sweep()` to the fee collector, not returned to the user.

Impact

Informational. Requires admin misconfiguration (allowlisting an invalid address) combined with a caller submitting a swap with insufficient `minAmountOut` protection.

Recommendation

Validate in `_setRouter()` that `router != address(0)` and `router.code.length > 0` when `allowed == true`.

Developer Response

Fixed in PR#656.

2.11 Final Remarks

The MultiSwapRouter handles a wide range of DEX integrations, which inevitably introduces complexity. The audit identified one medium severity issue and several low severity findings, primarily around swap leftovers, token validation, and step continuity — all of which were promptly fixed. The codebase is solid overall with limited attack vectors, and the development team was responsive throughout the review.