DECENTRALIZED INTELLIGENCE

Audit Report

2026-01-02

Produced for

AVEDEX OPT-FZCO

1. Scope

This security review covers the Solidity contracts in `AveSwapRouter-main.zip` (SHA256 digest: `74c406dc3f80f0186ae0d3f75562e9c8bc898f091137a82083b7c763c85e3fe6`):

- `contracts/AveBitRouterV4.sol`
- `contracts/AveBitLib.sol`
- `contracts/AveBitQuoterV4.sol`

2. Out of Scope

- Off-chain routing/quoting/pathfinding systems and any backend/UI
- Third-party protocol codebases (e.g., Uniswap/Pancake/Aerodrome), except integration usage in scope contracts

3. Executive Summary

The system is a path-driven swap router that can execute "v2-style", "v3-style" and "v4-style" swaps based on fields in `PathV4` (`poolId`, `router`, `pair`). The contracts assume a trusted off-chain path builder and a trusted, owner-controlled `quoter`.

The most severe issue is a **critical authentication bug in the Uniswap v3-style swap callback**: any address can call the router's callback functions and force the router to transfer arbitrary ERC20 tokens to the caller. Even if the router is intended to be "stateless", this implementation can hold balances (transiently during swaps, and persistently via dust/stuck funds), so this is a direct fund-theft primitive whenever balances exist.

Overall risk: Critical until the callback authentication issue is remediated.

Companion integration smoke test report. We also ran a fork-based integration smoke test to empirically validate common execution paths and failure modes across Ethereum, BSC and Base (pinned blocks; Top-50 tokens; `0.01` native buy→sell cycles). See the self-contained companion report in this deliverable: `smoke-test-report.pdf`.

4. System Overview (as implemented)

4.1 Swap description and path typing

`SwapDescriptionV4` includes a `paths` array of `PathV4` hops (`contracts/AveBitLib.sol:114`).

The code uses the following hop classification in `AveBitLib.swapDesc` (`contracts/AveBitLib.sol:772`):

- **v4**: `p.poolId != bytes32(0)`
- **v3-style**: `p.poolId == bytes32(0)` and `p.router == address(0)` (pool at `p.pair`)
- **v2-style**: `p.poolId == bytes32(0)` and `p.router != address(0)` (pair at `p.pair`)

4.2 Fund flow summary

- The router (`AveBitRouterV4`) receives the input from the caller:

  - ERC20 inputs are transferred via `TransferHelper.safeTransferFrom` in `_beforeSwap` (`contracts/AveBitRouterV4.sol:135`).
  - ETH inputs are received via `msg.value`.
  - If the first hop is v2-style and the input is an ERC20, the input is sent directly to the first pair (`srcReceiver = desc.paths[0].pair` in `contracts/AveBitRouterV4.sol:187`).
- `AveBitLib.swapDesc` executes each hop (`contracts/AveBitLib.sol:772`):
  - **v2-style hops**: the input must already be in the pair; the pair sends output to the chosen `recipient` via `pair.swap(...)` (`contracts/AveBitLib.sol:376`).
  - **v3-style hops**: the pool sends output to `recipient` via `pool.swap(to, ...)` (`contracts/AveBitLib.sol:434`), and then pulls the input from the router during the callback (router pays via `TransferHelper.safeTransfer(...)` in `contracts/AveBitLib.sol:879`).
  - **v4-style hops**: the router approves Permit2, Permit2 authorizes the Universal Router for a short window, and the Universal Router pulls inputs from the router and returns outputs (measured by balance delta) (`contracts/AveBitLib.sol:637`).
  - For gas efficiency, **v2 is the only hop type that requires inputs to be pre-positioned in the pair**, so `swapDesc` sometimes routes intermediate outputs directly into the *next v2 pair* (recipient = `desc.paths[i + 1].pair`) when the next hop is v2-style (`contracts/AveBitLib.sol:798`).
- Final output is delivered to `desc.dstReceiver` (or unwrapped to ETH when `dstToken` is the ETH sentinel).

5. Severity Definitions

- **Critical**: direct theft of funds or full compromise with no special conditions
- **High**: loss of funds or major break with realistic conditions / common integrations
- **Medium**: loss of funds in edge cases, or significant availability / incorrect behavior
- **Low**: minor risk, limited availability issues, or common compatibility issues
- **Informational**: best practices, clarity, and monitoring improvements

6. Findings Summary

ID	Severity	Title
F-01	Critical	Unauthenticated v3-style swap callbacks allow draining router ERC20 balances
F-02	Low	v4 ETH-out transfer uses `p.tokenOut` vs computed `outToken` (mostly latent; edge-case transaction failure/mis-payout)
F-03	Informational	Dust retention + owner token recovery are trust/operational considerations
F-04	Informational	Missing events for privileged changes and token recovery
F-05	Informational	Non-standard `ReentrancyGuard` initialization
F-06	Informational	`slippage` parameter is always `0`; only `minReturnAmount` is enforced
F-07	Low	`safeApprove` does not handle USDT-style `approve(0)` and has allowance race considerations

7. Detailed Findings

F-01 (Critical): Unauthenticated v3-style swap callbacks allow draining router ERC20 balances

Affected code

- Router exposes public callback entrypoints:
  - `AveBitRouterV4.uniswapV3SwapCallback` (`contracts/AveBitRouterV4.sol:249`)
  - `AveBitRouterV4.pancakeV3SwapCallback` (`contracts/AveBitRouterV4.sol:257`)
  - `AveBitRouterV4.algebraSwapCallback` (`contracts/AveBitRouterV4.sol:266`)
  - `AveBitRouterV4.hyperswapV3SwapCallback` (`contracts/AveBitRouterV4.sol:274`)
- All callback entrypoints forward to `AveBitLib.v3SwapCallback` (`contracts/AveBitLib.sol:879`).
- `AveBitLib.v3SwapCallback` transfers `data.tokenIn` to `msg.sender` with no caller authentication (`contracts/AveBitLib.sol:892`).

Issue

In Uniswap v3-style swaps, the pool calls the recipient contract's callback and expects the contract to pay the input token. This router implements the callback but **does not verify that `msg.sender` is the expected pool** for an in-progress swap.

As a result, any account can call the callback function directly and cause the router to transfer arbitrary ERC20 tokens to the caller.

This is precisely the class of issue Uniswap's official periphery router defends against via callback validation (computing the expected pool address from a trusted factory + pool key and requiring `msg.sender` equals that pool). This implementation has no analogous check.

Impact

Any ERC20 token balance held by the router contract can be stolen, including:

- dust accumulated across swaps (see F-03),
- tokens accidentally transferred to the router address,
- tokens temporarily held between hops (if a malicious external contract can reenter into the callback during swap execution).

Reproduction (minimal)

1. Ensure the router holds some ERC20 token `T` (e.g., `T.transfer(router, amount)`).
2. Call the callback directly:
   - Choose `tokenOut` so the address ordering condition passes:
     - If `T < tokenOut`, use a positive `amount0Delta`.
     - If `T > tokenOut`, use a positive `amount1Delta`.
   - Encode callback data as `abi.encode(tokenIn, tokenOut)` (two addresses).
3. Observe the router's `T` balance decreases and the caller's `T` balance increases.

Remediation recommendations

Implement strict callback authentication. Standard approaches:

- **Store the expected pool address in storage** immediately before calling `IPairV3(pool).swap(...)`, clear it immediately after, and in each callback entrypoint require `msg.sender == expectedPool`.

- Additionally bind the callback to expected token(s) (token0/token1) or expected hop metadata (e.g., store a hash of `(tokenIn, tokenOut, pool)` for the active hop).
- Avoid relying on user-provided callback data for authentication; it is attacker-controlled.

Until fixed, the router should be treated as unsafe to hold balances.

F-02 (Low): v4 ETH-out transfer uses `p.tokenOut` vs computed `outToken` (mostly latent; edge-case transaction failure/mis-payout)

Affected code

- `_swapV4` sets `outToken = ETH` when `paramEx.isEthOut` (`contracts/AveBitLib.sol:697`).
- `_swapV4` then transfers `p.tokenOut` instead of `outToken` (`contracts/AveBitLib.sol:705`).

Issue

For v4 pools involving native ETH, `_swapV4` measures output by tracking the router's ETH balance delta and sets `outToken` to the ETH sentinel. However, the transfer uses `p.tokenOut`.

In the current execution flow (`swapDesc`), v4 hops usually pass `to = address(this)` (intermediate hops always do; and for a final ETH output the router also keeps output in-contract so it can send ETH at the end). In those common cases, `safeTransfer(..., to, ...)` is a no-op because `to == address(this)`, so this mismatch is typically latent.

It becomes observable if a route builder encodes a **native-ETH v4 pool** but represents the ETH leg as **WETH** in `p.tokenOut` while also setting a non-contract recipient (e.g., last hop with `desc.dstToken != ETH`), or otherwise produces a situation where `_swapV4` computes ETH output but then attempts an ERC20 transfer.

Impact

- Route-specific transaction failure for inconsistent ETH/WETH encoding (swap reverts when attempting to transfer `p.tokenOut` even though the swap produced native ETH).
- In edge cases where the router holds balances of `p.tokenOut`, potential mis-payout (pays `p.tokenOut` while retaining native ETH output internally).

Remediation recommendations

- Transfer using `outToken` (the computed output token) rather than `p.tokenOut`.
- Alternatively enforce a strict convention: require that `p.tokenOut` is the ETH sentinel whenever `paramEx.isEthOut` is true, and revert otherwise.
- Consider explicit wrapping behavior if the intended UX is "WETH out" rather than "native ETH out".

F-03 (Informational): Dust retention + owner token recovery are trust/operational considerations

Affected code

- `_refundIfStuck` only refunds "stuck" input tokens if the leftover is at least 5% of `amountIn` (`contracts/AveBitLib.sol:865`).

- The router owner can withdraw arbitrary tokens/ETH from the router via `recoverTokens` (`contracts/AveBitRouterV4.sol:88`).

Issue

The router intentionally retains small leftover balances (dust) for v3/v4 hops, and the owner has the ability to withdraw any retained assets.

This may be an intentional business decision, but it is a trust/operational consideration that should be disclosed to integrators and users. It also increases the impact of F-01 because retained balances become directly stealable.

Impact

- Users may lose small residual balances (dust) across swaps.
- Owner can capture retained balances; monitoring is harder without events (see F-04).

Remediation recommendations

- Refund all leftover balances deterministically, or make the dust threshold configurable and transparent.
- Emit an event on token recovery and privileged parameter changes (see F-04).
- If retained balances are intended as protocol revenue, document the policy and recipient(s).

F-04 (Informational): Missing events for privileged changes and token recovery

Affected code

- `setOwner` and `setQuoter` do not emit events (`contracts/AveBitRouterV4.sol:74`).
- `recoverTokens` does not emit an event (`contracts/AveBitRouterV4.sol:88`).

Impact

Operational monitoring and incident response are harder without events, especially given the owner's ability to change the `quoter` and withdraw assets.

Remediation recommendations

- Emit events for `OwnerChanged`, `QuoterChanged`, and `TokensRecovered`.

F-05 (Informational): Non-standard `ReentrancyGuard` initialization

Affected code

- `_status` is never initialized to `_NOT_ENTERED` (`contracts/AveBitRouterV4.sol:21`).

Impact

This is not a direct vulnerability (the modifier still prevents reentrancy), but it is a divergence from the canonical OpenZeppelin pattern and can affect gas refund behavior on the first call.

Remediation recommendations

- Initialize `_status = _NOT_ENTERED` in a constructor.

F-06 (Informational): `slippage` parameter is always `0`; only `minReturnAmount` is enforced

Affected code

- `swapV4` calls `_executeSwap(d, 0, SwapMode.ROUTE)` (`contracts/AveBitRouterV4.sol:105`).
- `exactOutV4` calls `_executeSwap(d, 0, SwapMode.EXACT_OUTPUT)` (`contracts/AveBitRouterV4.sol:112`).
- The `OVER_SLIPPAGE` check is gated behind `slippage > 0` (`contracts/AveBitRouterV4.sol:222`).

Issue

The router contains a slippage-based check against a pre-swap quote (`quoteOut`), but the public entrypoints always pass `slippage = 0`. As a result, the slippage-based check is effectively disabled in all current flows, and the only always-on execution protection is `amountOut >= desc.minReturnAmount` (`contracts/AveBitRouterV4.sol:228`).

Impact

- Integrators may incorrectly assume the router enforces an additional slippage bound against `quoteOut`; in practice they must set `minReturnAmount` appropriately.

Remediation recommendations

- Either remove the unused `slippage` parameter and dead branch, or expose a public entrypoint that accepts a slippage value (and document its semantics relative to `minReturnAmount`).

F-07 (Low): `safeApprove` does not handle USDT-style `approve(0)` and has allowance race considerations

Affected code

- `TransferHelper.safeApprove` (`contracts/AveBitLib.sol:6`).
- `_swapV4` approves `permit2` for `inAmount` via `TransferHelper.safeApprove` on every v4 hop (`contracts/AveBitLib.sol:667`).

Issue

`safeApprove` performs a raw ERC20 `approve(spender, value)` call and assumes success with either no return data or an ABI encoded `bool`.

This has two common integration pitfalls:

- **USDT-style approvals**: some tokens require setting allowance to `0` before setting a new non-zero allowance. Repeated swaps can fail if the router tries to update a non-zero allowance directly.
- **Allowance update race**: the well-known ERC20 allowance race when changing a non-zero allowance to another non-zero allowance can allow the spender to use both the old and new allowance if the spender acts between transactions. While `permit2` is typically treated as trusted infrastructure, this remains a best-practice and documentation consideration.

Impact

- v4 swaps can revert for tokens that enforce `approve(0)` first.
- If the approved spender is untrusted, allowance races can lead to overspending relative to the intended allowance update.

Remediation recommendations

- Use a force-approve pattern: if `approve(value)` fails, call `approve(0)` and then `approve(value)`.
- Consider approving a large allowance once (and only increasing when needed) to avoid frequent non-zero→non-zero updates.
- If possible, prefer `increaseAllowance/decreaseAllowance` style updates (or OpenZeppelin `SafeERC20.forceApprove`) and document the spender trust assumptions.

8. Closing Notes

This router is a flexible, path-driven executor across multiple AMM styles. The security posture is therefore dominated by two factors: (1) the correctness of low-level token/payment invariants during execution, and (2) the safety of external-call surfaces (pools/routers/callbacks).

The top priority remediation is to harden the v3-style callback surface (F-01). Uniswap's official periphery routers explicitly validate callbacks by requiring `msg.sender` to be the expected pool address computed from trusted pool parameters. This implementation currently lacks any equivalent check, which enables direct ERC20 fund theft whenever the router holds balances (including transient balances during swaps, residual dust, or accidental transfers).

Separately, the report highlights several semantic/operational items that should be made explicit to integrators: the ETH/WETH representation conventions for v4 paths (F-02), the fact that the "slippage vs quote" branch is currently unused because `slippage` is always passed as `0` (F-06), and the privileged/operational surface (token recovery and configuration changes) which benefits from transparent eventing (F-03/F-04), and v4 approval/allowance handling for `permit2` (F-07). If this contract is intended for broad public use, addressing these items will materially improve safety, debuggability, and integration correctness.

Disclaimer

This report was created on: January 6, 2026. We hope you find this report informative and useful. If you have any questions or need further clarification, please do not hesitate to contact contact@d23e.ch.

The report is provided solely for informational purposes and should not be considered as an endorsement, recommendation, or any form of legal, financial, or investment advice.

The report is based on the code at the time and does not account for any updates, modifications, or alterations to the code that may occur after the report date. The code was assessed "as-is" and the findings represent the state of the code at the time of the assessment.

Although every reasonable effort has been made to ensure the accuracy, completeness, and fairness of the report and findings contained within the report, it is provided on an "as-is" basis without any warranties, representations, or guarantees of any kind, express or implied. This includes, but is not limited to, warranties of merchantability, fitness for a particular purpose, non-infringement, accuracy, or the presence or absence of errors, whether or not discoverable.

The authors, evaluators, and any associated parties disclaim all liability for any losses, damages, costs, or expenses (including legal fees) arising directly or indirectly from the use of or reliance on the report or its findings. This includes, but is not limited to, any damage or loss caused by errors, omissions, inaccuracies, or any misleading or out-of-date information.

The reader is solely responsible for any actions or decisions taken based on the information provided in this report. It is highly recommended that, where necessary, appropriate professional advice is sought before making any decisions or taking any actions relating to the smart contract code analyzed in this report.

We would like to reiterate that the report is no replacement for a real, comprehensive audit involving significant manual labor.